5)

A Distributed Multimedia Repository: Access Protocol Design and Implementation

by

Scott David Centurino

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

Author .....
Department of Electrical Engineering and Computer Science
May 30, 1995

Certified by .....
Steven R. Lerman
Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Theses

A Distributed Multimedia Repository:
Access Protocol Design and Implementation

by

Scott David Centurino

Submitted to the Department of Electrical Engineering and Computer Science
on May 30, 1995, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The thesis describes the design and prototype implementation of a protocol for accessing a distributed multimedia repository. The protocol is based on a model of a multimedia repository in which generally unstructured media data is stored in a distributed manner among an arbitrary number of host machines which form the repository, while structured data related to each element archived in the repository is stored in a formal database. The access protocol specified provides a uniform view into this model, integrating the media data from the repository into the logical database while it remains external to the physical one.

The prototype *MediaServer* implements the protocol within an architectural model which extends the model specified in the protocol. The prototype server is functional and provides full access to a test database and its associated repository of images. The prototype implementation also provides a C++ class library to allow application clients to be written easily. A critical element of the prototype is the object system on the server which supports the exporting of local server objects to the connected client, and the corresponding object system on the client.

Thesis Supervisor: Steven R. Lerman
Title: Professor of Civil and Environmental Engineering

Acknowledgments

The prototype *MediaServer* described herein builds upon the efforts of many people to whom I am eternally grateful. I would like to thank the AM2 development staff for their assistance and cooperation. I would like to especially thank the people responsible for the four pillars upon which the *MediaServer* prototype is built. Jud Harward is responsible for Utility layer of AM2 which provides the UTlist and UTvalue mechanisms which are so critical to the server. Masanori Kajiura (Kaji) and Tomi Tominaga built the AM2 network layer. Kaji designed and implemented a custom network and process control layer specifically for my use in the *MediaServer*. Finally, Kate Curtis designed and implemented the current version of the AM2 database classes, based on an original design by Steve Lerman.

I would like to thank Steve Lerman, Jud Harward, and Phil Bailey for their extensive input into the design of the original image server architecture, upon which the *MediaServer* architecture is based. Refinements of the low level system were heavily influenced by Kaji, Kate, and Tomi. I would like to further thank Professor Lerman for his help preparing this document and in general for agreeing to supervise my thesis.

I would like to express my eternal gratitude to Kate Curtis and Adam Feder. Kate's selfless commitment to doing extra work to support my thesis was never-ending. Adam was my tireless sounding board. I am convinced that without his input and insight into issues ranging from the object system to the formatting of this document, and his assistance in building on top of the AM2 C++ classes, I never would have finished. I only hope that I can ever repay either of these debts.

On a more personal level, I'd like to thank all my housemates for their patience and understanding during the last two terms, especially as I neglected more and more things around the apartment. My sanity was in their hands, and I think it has mostly survived. Of course, my sanity and health were most directly saved by my loving fiancé, Victoria Scheribel. For all her support in everything I do, and everything I don't, I can only say thanks. Luckily, I get to keep saying it for the rest of my life.

Lastly, I would like to thank my parents for all of their support, both financial and psychological, over the last 24 years. They have made many sacrifices to offer me the ability to pursue my dreams and ambitions. This thesis, and all the work it represents, is dedicated to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The storage and access of unstructured data is a difficult task. Conventional database management systems are designed to handle "facts": inherently structured data elements which conform to specific database schema. Increasingly, multimedia applications are being developed which need to access a full range of data types including full text, audio, video, and image. Traditional databases attempt to deal with this need by treating large data items, such as articles, pictures, sound bites, *etc.* as single records of an indeterminate **B**inary **L**arge **OB**ject type (BLOB) stored within the database. This design is not very efficient, nor is it nearly as useful to the user of the database as it could be.

This thesis documents the design and a prototype implementation of a protocol for accessing large amounts of multimedia data. The MediaServer protocol relies on a model of data storage which couples a repository of unstructured data (*e.g.* media elements stored in files, but with no inherent ordering of the files) with a database of related, structured data. The underlying goal of the MediaServer protocol is to provide a uniform front end to this model, through which an application accesses both the database and the repository. This allows for the logical consistency required to make the server appear a single information source to client applications interfacing with it, while allowing the database to be optimized for structured data and the repository to store the unstructured media data in an efficient, distributed manner.

Until recently, research tended to be focused on specific systems optimized for the

types of data they contain [1]. These systems have generally been custom designed, from the user interface down to the storage techniques (see for example [2]). With the increasing popularity of object-oriented techniques and databases, this trend has shifted a bit, since systems can be designed using classes of data, and specific data bound to the classes later. These systems still tend to be implemented as complete applications. Only very recently has development effort gone into designing general-purpose media servers which can be accessed by a variety of clients [3, 4]. The *MediaServer* project falls into this latter class.

This thesis has two major components. The first is the specification of the base architectural model of a MediaServer and the protocol for accessing that model. The second is a prototype implementation of a server conforming (for the most part) to the specified protocol. Throughout this thesis, the terms "MediaServer" (unitalicized) and "MediaServer protocol" refer to the protocol and/or its model. The terms *MediaServer* (italicized) or "*MediaServer* project" refer to the prototype implementation, its architecture, or this thesis project as a whole.

1.1 History

MediaServer grew out of a project at the Center for Educational Computing Initiatives (CECI) called the Image Server. This project was initially motivated by the desire to create an electronic version of the Rotch Visual Collection kept by the MIT libraries. This large collection (the bulk of which is an archive of approximately 300,000 35mm slides) is used primarily by faculty and students in the School of Architecture and Planning.

I was heavily involved in the original Image Server project, designing the original system's architecture. Professor Steven R. Lerman was also heavily involved in the project, guiding me in the architectural design and developing a generic database access model called the Virtual Database Model. The Image Server would provide access to the repository's database using the interface specified by the Virtual Database Model. Each Image Server would be required to have an implementation of the Vir-

tual Database interface which provided access to the underlying database.

The project eventually evolved into the current *MediaServer* project when the Virtual Database Model was divorced from the Server. One major reason for this was that it became readily apparent that the Virtual Database was a subsystem which was clearly useful in its own right. Furthermore, the design issues of the Virtual Database were for the most part independent of the *MediaServer*. The Virtual Database project has been undertaken by another student, Katherine A. Curtis. A pair of visiting scientists, Masanori Kajiura and Tomi Tominaga, offered to provide a network interface layer. I concentrated on evolving the architecture and access protocol of the system.

The *MediaServer* project is not attempting to provide a way for users to access, browse, *etc.* given repositories, but rather it specifies a protocol which provides a way for *programmers* to easily write applications which provide this kind of access.

1.2 Scope

The scope of the thesis includes the design of the model architecture, the access protocol, and the prototype implementation architecture, and the prototype implementation itself. The protocol specified includes only "read" capability. The issues involved in supporting write capability, including security and coherency issues, were considered beyond the scope of this thesis.

Issues related to data transport are not addressed. Instead, *MediaServer* uses an existing library of code which handles data format exchange and efficiency issues.

1.3 Design Goals

The major design goal of the overall *MediaServer* system is to integrate the view of a database and large media object files externally stored in a possibly distributed file system. A secondary goal is to provide efficient and timely access to elements stored in either data storage system. This section provides details about the resulting goals

for the protocol and for the protocol implementation.

1.3.1 Protocol Design

The primary goals (P*n* in the following lists) of the MediaServer protocol can be broadly described as programmability, scalability, and extensibility. Related secondary goals (S*n* in the following lists) include performance, efficient use of limited database resources, and security.

Programmability in the case of MediaServer deals with the ease of writing programs to access repositories. It leads to two primary system requirements, and one secondary requirement:

P1 The client application programmer should be dealing with the simplest view of the system possible. MediaServer's responsibility is to provide a *simple, unified view* of a complex system to allow the application programmer to be free from the responsibility of learning how to coerce a complex system to provide the access he needs. The client interface to the protocol should naturally lead to an implementation as a client Application Programming Interface (API).

P2 The programming interface to this simple view should be *minimal* and as *error-reporting* and *fault-tolerant* as possible, allowing the application programmer to decide how to deal with errors.

S1 Decisions about user interface, media presentation, and when to transport objects (especially media) should be up to the application programmer whenever possible.

The scalability requirement is fairly straightforward:

P3 The system should be capable of accommodating a significant number of concurrent users (at least dozens) accessing a media collection of arbitrary size (at least millions of distinct media elements).

Extensibility provides two more primary requirements:

P4 The protocol should be designed in such a way that the specification can be augmented and/or modified with minimal impact on implementations compliant with older specifications.

P5 The protocol should be designed in such a way that protocol implementers have as much latitude as possible to optimize their particular implementations. Implementations should be able to provide specific customized capabilities whenever this makes sense (such as supporting stronger authentication, or providing server-side media manipulation capability).

The performance goals include the implied goal of reasonable performance when the system grows, as well as:

S2 The protocol should impose as little overhead on the transport of media elements as possible.

Resource usage is an important secondary goal which affects performance:

S3 The protocol should impose as little overhead as possible on database resources.

Finally, security issues, while not a prime concern, needed to be addressed:

S4 The protocol should provide reasonable security mechanisms.

Observations

At the root of almost any problem in computer science is the tradeoff between flexibility and performance. In every portion of the design of this system, flexibility has come first. Of course, within the flexibility constraints, effort has been made to improve the performance of the system. It is important to recognize that the performance goal for the project was to provide at least adequate performance. It was never a goal to provide a high-performance system, because doing so was seen as fundamentally at odds with the primary goals of programmability and extensibility.

1.3.2 Prototype Implementation

Obviously, the prototype implementation is intended as a proof-of-concept for the MediaServer protocol. In fact, the term prototype may actually be an inaccurate description of the Server and Client API described in Chapters 3 and 4, since they are in fact functional enough to be used as a full implementation.

Of paramount concern in the first implementation was the requirement that making the *MediaServer* accessible from the Application Description Language (ADL) of the AthenaMuse® 2 Multimedia Authoring Environment (AM2) be simple. AM2 is the major effort at CECI. It was this accessibility requirement which motivated the object-oriented design of the Client API, as well as the use of the UTlist and UTvalue facilities of AM2. The MediaServer prototype must use the AM2 network layer in order to achieve this accessibility.

One major positive effect of the need to conform to AM2 is that it was very natural to use the aforementioned facilities in the Server itself, not just in the Client API. It was very natural to use the AM2 implementation of the Virtual Database Model on the Server. It was also important to make accessing the MediaServer through the Client API as similar to accessing the simpler Virtual Database API as possible.

The prototype's architecture was designed for maximum robustness and for maximum efficiency in the use of major system resources, especially database licenses. Areas which were given less attention include security and scalability of user access.

The number of concurrent users is does not scale as well as it should, as performance degrades faster than it should as users are added. Instead of being limited by the capabilities of the underlying database, the number of users and performance is limited by the fact that all connections are handled by the same machine. The protocol is designed to accommodate implementations which maintain sessions on multiple hosts, but in the first implementation there is a single host acting as the initial contact point. The process which provides the contact point is only capable of creating sessions on the connect host.

1.4 Structure of this Thesis

The rest of this document is organized as follows. Chapter 2 provides the MediaServer protocol specification, including a description of the model upon which the protocol is based. Chapter 3 describes in detail the prototype implementation of the server side of *MediaServer*. The client API provided for the prototype *MediaServer* is presented in Chapter 4. Chapter 5 presents a short evaluation of the protocol's performance, and concludes by suggests several future directions for *MediaServer*.

Chapter 2

Protocol Description

This chapter serves as a specification of the initial version of the MediaServer protocol. Section 2.1 introduces the models of the system upon which the protocol relies. There are actually two models of the system which are relevant to the MediaServer. Which is the appropriate one to use depends on the level of the protocol being examined. Section 2.2 describes the required formats for messages among the system components. Section 2.3 specifies the MediaServer commands available to the Client, and Section 2.4 specifies the valid replies from the server. Finally, Section 2.5 specifies how future specifications of the protocol should be assigned version numbers.

2.1 The MediaServer Model

There are two models important to the MediaServer protocol: the simple model which should be apparent to a programmer, and the more complex, actual model of system communication. The simple model is the one on which the protocol commands are based, while the full model is necessary to specify the actual actions taken as a result of those commands being invoked.

2.1.1 The Simple Model

Figure 2-1 presents a diagram of the simple model. In this model, the application invokes the MediaServer client, which initiates the Command connection. At the

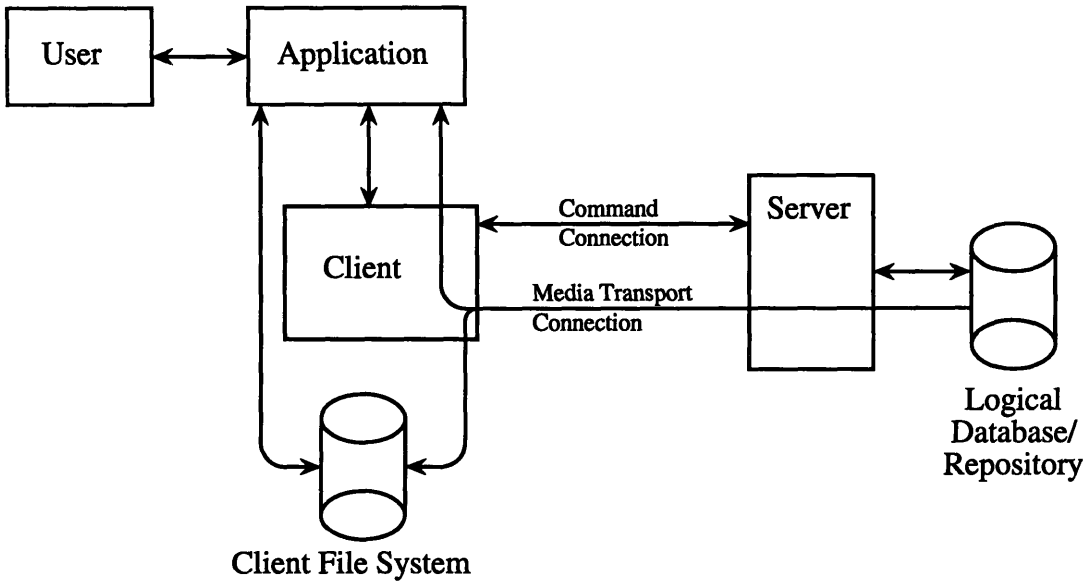

Figure 2-1: The Simple MediaServer Model

initiation of the application executed by the user, the MediaServer client generates MediaServer command messages and transmits them to the server process via the Command connection. Reply messages are sent over the Command connection in response to the command messages.

Commands can be broken into three categories: connection commands, database access commands, and media request commands. Connection commands are used to initiate or terminate a connection between the client and the server. They also provide a facility for the server to limit the number of concurrent active connections and to maintain a queue of pending connections from which to take new active connections.

Database access commands conform to the Virtual Database Model. The Virtual Database Model is an abstract, *object-oriented* representation of a database which provides a uniform interface to any underlying logical database. The fact that the Virtual Database is object-oriented makes the access protocol of the MediaServer inherently *object-based*, meaning that all non-connection commands are invoked on specific objects in the system. More detail on the implications of an object-based model is presented in Section 2.1.3. Database access actions through the Virtual Database Model include query execution, cursor (query result) access, database object

attribute retrieval and method invocation.[1]

Media request functions are used to request the manipulation and/or sending of the binary media data associated with a specific media object on the server. Media requests utilize a second connection stream between the server and the client. When the client requests a media element, the server prepares a connection and replies with information about that connection. The client then initiates a Media Transfer Connection based on that information, and uses it to fetch the binary media data.

2.1.2 The Full Model

The full model of the MediaServer protocol is diagrammed in Figure 2-2. The major addition to the model is that the complexity of the server is revealed. The logical database from the simple model is now viewed in terms of its physical components, namely the physical database and the external media storage components, or "Repository components," which are referred to as a whole as "the Repository." Each Repository component is defined to be a single machine and its local data storage. It is accessed through one or more dedicated process(es) called Retrievers, each of which is capable of maintaining independent Media Transfer Connection(s) with client(s). The number of Retrievers is only specified to be at least one per Repository component. It is up to the implementor to determine whether a single Retriever per component is sufficient, or if there will be a Retriever for each client.

The Command connection is maintained by a process called the MediaServer Element Request Information Server (MERIS). The protocol does not specify the communication architecture between Retrievers and MERIS's. Again, the protocol does not specify the number of MERIS's; there can be a MERIS per client, or there can be a single MERIS for all clients.[2]

Separate Retrievers are run on each Repository component for various reasons.

[1] As discussed in Section 1.2, the current protocol supports only read operations and contains no write semantics.

[2] An architecture with a single MERIS will not scale well, since that MERIS is the single point of contact for the entire system, and every single request involves it. In the case of the Retriever, a single process per Repository component is feasible since it is unlikely that every client will be accessing every Repository component with anything greater than moderate frequency.

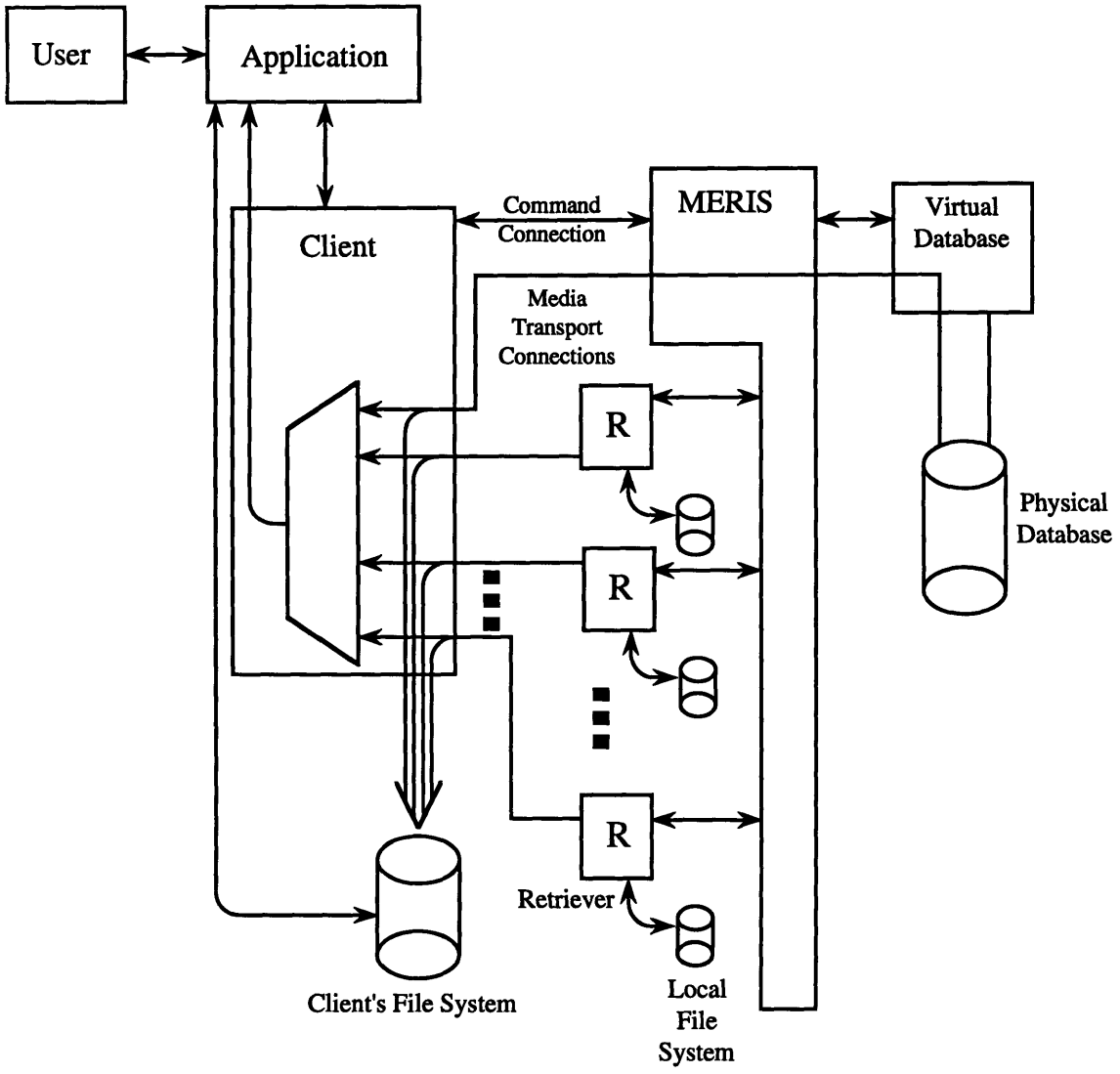

Figure 2-2: The Full MediaServer Model. The boxes containing **R**'s denote Retrievers

The first is that there is no guarantee that there is any direct file access between any pair of components or even between a component and the host running the MERIS. The second reason is to improve efficiency. If some kind of direct access is available, such as through a networked file system, it is possible to run only one Retriever, having it access media elements through the file system and send them to the client. This is quite inefficient, as it requires a minimum of two transfers of the media. One is the desired transport from server to client. The second, unwanted transmission is from the file server storing the data to the Retriever process's machine. The third reason for running separate Retrievers is that doing so improves the robustness of the system. If one Retriever fails, clients are affected only if they need that particular Retriever to access their current media request(s).

When the MERIS receives a media request command, it determines which Repository component contains that media and contacts/creates a Retriever responsible for providing access to that component. The MERIS and the database in combination are responsible for providing any access control deemed necessary. If there is not an active connection already in place for this client, the Retriever "listens" on a port for a client connect. The MERIS returns to the client the host address and port of the Retriever. The client then makes contact with the Retriever. As soon as the connection has been made, the Retriever begins sending the data. The client either immediately begins to read that data into a local file (denoted in the figure by the connections from the Media Transfer Connection to the client's file system), or it passes a handle to the stream (minimally through a file descriptor) to the application. The multiplexor shown in the client in the figure is to indicate that only one active stream is passed to the client at a time. The base protocol specifies that this stream must be valid (subject to network failure) until at least the next media request is initiated on behalf of the application. Implementations may choose to strengthen this guarantee, and thus effectively remove the logical multiplexor from the model.

It is possible for the physical database to contain media directly. In these cases, the MERIS may choose to act as a Retriever and set up a second connection as a Media Transfer Connection. This Connection must be no different than any other

Media Transfer Connection in the system. Specifically, the MERIS must guarantee that the Command connection is still active while it is performing the media transfer, so that the client may request that the transfer be aborted.

The interaction between the MERIS and the Retriever is explicitly not specified, nor is the operation of the MERIS upon startup.

2.1.3 An Object-Based Protocol

Almost all interactions with a MediaServer, other than the functions for initiating a connection and terminating it, are through methods invoked on objects. This is mainly due to the use of the Virtual Database Model, which is *object-oriented*. However, the protocol itself is not inherently object-oriented, a paradigm which includes such mechanisms as polymorphism and inheritance. Rather, the protocol is *object-based*, meaning only that the data on the Server is stored in and accessed through objects.

Although object terminology is anything but standard, the following definitions are sufficient for reading and understanding this specification. An object is a collection of data which can be treated as a single entity. Creating an object is also known as *instantiating* the object. Objects can have functions called *methods* invoked "on" them, meaning that the function has automatic access to the object on which it was invoked. A *class* is a definition of the structure of a type of object and the methods available to objects of that type. Thus, each object is an *instance* of the class which defines its structure and methods. A class's methods can only be invoked on objects of that class. Finally, in most cases, the terms class and type can be used interchangeably. Finally, in most cases, the terms class and type can be used interchangeably.

Using objects introduces several features useful to a MediaServer. One of the most important is a natural way to delay shipping of information. Imagine a complicated query which returns 100 records, each with 30 entries, including 3 images each. Clearly it is undesirable to have the MediaServer return *all* of this information immediately. The Virtual Database Model uses an object called a *cursor* to represent the return

values of a query. The cursor is an object which allows a programmer to iterate over the records returned by the query, returning them to the client one by one. The use of objects also provides a natural way to allow the client to represent media elements as abstract entities on the server, but still operate on them locally without requiring retrieval of the element to the client. The only alternative is to supply the filename of the element as stored on the server. In addition to eliminating the ability to provide "local" operations which are carried out remotely, this also compromises the security goals of the system. A filename might allow access to the element though some means other than the authenticated MediaServer.

An explicitly object-based protocol is not that different from other protocols. In a very real sense, an object structure provided by the MediaServer acts very much like the directory structure of a file system being accessed through the File Transfer Protocol (FTP). Each object contains a series of attributes, which can be either other objects or protocol base types (String, Integer, *etc.*). These attributes are very similar to a directory containing either other directories or files. The one additional feature of an object is that it has *methods*, which can be invoked on the object. Methods provide a standard way for an object to expose custom interfaces. Methods have no file system analog.

Requirements for support of objects

Even if the client is not built using an object-oriented or object-based language, working with objects is still not difficult. All objects in the protocol represent base Virtual Database Model "classes" (including media elements) or represent custom classes of the particular database.[3] These classes are part of the specification (see Section 2.2.2), as is the information sent by the server when an object is "sent" to the client. For most objects, it is possible to extract all the information they contain without incurring additional network overhead, since all of their information is exported with their reference. The client can then represent these objects in data

[3]This second type of object is actually one of the base types of the Virtual Database Model, but it is so different from the others that it warrants separate treatment.

structures containing their exported information, and can chose to store the object ID's or ignore them. The remainder of objects require communication with the Server to access information. The best way to provide such an object in a non-object-based environment is to again use a data structure to represent the object, but this time the object's ID must be stored, so that it can be supplied to protocol commands when the object is accessed.

2.1.4 Communication Layer Assumption

The protocol outlined in this chapter requires the use of some form of reliable communication capable of transporting both binary and text data over the Command and Media Transport connections. TCP/IP is probably the most likely and best candidate since it is a standard reliable communication protocol. The transport facility must provide solutions to cross-platform communication issues, such as byte order, byte/word size, etc, as the protocol does not address these issues at all.

2.2 Message Format

Specifying the *transmission format*—the exact format of the data of a message placed onto the communication stream—is beyond the scope of this thesis. Instead, the message format is specified in terms of an abstract list utility called a List. A List is an ordered sequence of Values. A Value can contain exactly one datum, which can be of any of the following data types: Integer, Real, String, Boolean, ObjectID, List, Object, Binary, or Unset. The Object type is a reference to the memory containing an object on the server. An Object can never be transported across a communication channel because the physical reference has no meaning to the client. Instead, an ObjectID which uniquely identifies that Object is sent. The Binary type is intended to allow encrypted data to be easily sent in a message. The Unset "type" has no data associated with it, but instead acts as a placeholder for elements which are required but for which no data is available. Unset does not need to be an explicit type, but can rather be a flag on a Value indicating whether it contains valid data. Note that

in a specific implementation, any of these types may be represented by the others. For example, a Boolean could have the typical Integer representation with False=0 and True=1.[4]

2.2.1 Transmission Format "Specification"

To specify the message transmission format given a abstract message List, it is sufficient to introduce two functions, *Encode* and *Decode*. Given an arbitrary message List $Message$, these two functions must have the following property:

$$Decode(Encode(Message)) = Message$$

This is sufficient because the requirement of a reliable, non-content-transforming transport layer has already been introduced.

In the future, the actual transmission format may be specified. Given such a specification of the transmission format, the requirements of *Encode* and *Decode* are strengthened. Under the specification, a particular $Message$ would have exactly one correct transmission format, $SpecMessage$. *Encode* and *Decode* then must possess the following properties:

$$Encode(Message) = SpecMessage$$

$$Decode(SpecMessage) = Message$$

Finally, without specifying the transmission format, it is still possible to allow multiple List implementations, by providing a pair of translation functions. Given a message, $Message_A$, formatted in one implementation and $Message_B$, the same message formatted in the second implementation, two functions $TransformAB$ and $TransformBA$ allow for the following transformations:

$$TransformAB(Message_A) = Message_B$$

[4]The minimal implementation would have every "Value" represented as a List containing two elements (a pair). The first element is a String containing the Type name of the second element. The second is either a List (if the type is List) or a string representation of the data.

$$TransformBA(Message_B) = Message_A$$

Although it is clearly better to specify an intermediate transmission format to or from which any List representation can be transformed, this second method involving only two different representations is fairly good, since only one side of the service needs to be aware that there are two representations.

2.2.2 Format of the Message List

The message list format is an open-ended part of the protocol. Every message list must have a minimum of two Values. In communication from the client to the server, the first Value is an Integer identifying the command to perform; the second is a List of the arguments to that function. In communication responses from the server to the client, the first Value is an Integer identifying the error code of the response; the second is a List containing the actual response, if any. Every message from the client to the server sent after an active connection is established is required to have a third Value: a List (called the *Kill List*) containing the ObjectIDs of any objects deleted from the client since the last time the server was sent a message. Additional Values can be added to the message list as required to support future protocol extensions.

To facilitate the "transport" of Objects, the structure of any List sent has to be altered. This alteration is called *marshalling* the list. Every List is marshalled into a new List of two Lists. The first contains all of the elements of the original List, except that every Object is marshalled into a List containing at least three elements, an ObjectID, an Object Type Identifier, and a List containing any immediately available data from the object. The second List in the marshalled representation of a List contains the Integer locations of all sub-Lists which need to be unmarshalled into Objects.

Currently, the Protocol directly supports the eight base object types of the Virtual Database: Date, Time, Timestamp, Monetary, Object, Cursor, Set, and Media. The first four have all of the information they contain transported in the data List. A Database Object has its name transported, but all methods on it must be performed

by the server. Likewise, a Cursor is shipped with all of its state information, which the Client is responsible for maintaining, but the method to access the records in the Cursor requires commands being executed by the server. A Set is the only Object type, other than a Cursor, which can potentially contain other Objects. Thus, the data List of a Set must be recursively marshalled if it contains any objects. The Media type is a base type which contains enough information to determine what type of media (image, video, audio, fullText) it contains. In object-oriented parlance, Media is a base class for classes representing the different types of media. This means that a Media object contains all of the general information a media object needs, while the media object itself, such as an Image object, contains both information specific to that type and a way to access its associated Media object.

There are various optimizations of this representation, such as the one in the prototype implementation in which all top-level lists become three lists, the third one indicating which Values of the first list are Lists which contain marshalled Objects. All other Lists can be safely assumed to contain no objects.

Data List Specifications

Each comma separated list below indicates the minimum information which must be supplied in the third List of a marshalled Object. Notice that both Cursor and Set have two possible forms, depending on the value of their first arguments.

Date: Integer month, Integer day, Integer year

Time: Integer hour, Integer minute, Integer second

Monetary: Real amount, String currency type

Timestamp: Integer month, Integer day, Integer year,
Integer hour, Integer minute, Integer second

Set: Boolean objects=True List <marshalled Objects>
Boolean objects=False List Values

Object: String name

Cursor: Boolean success=True, Integer row count, List type names
Boolean success=False

Media: String type (Image, Video, etc.), <type specific data list>

2.3 MediaServer Access Commands

The MediaServer protocol has two layers of commands which mirror the two levels of the model. The lower layer of commands has knowledge of the existence of Retrievers and manages connections to them for the upper layer.

In the lists of commands that follow, the upper layer commands are listed first and are capitalized and in normal font. The low level commands are listed after them and are in `typewriter` font. Any arguments required by a command are listed in italics after the command name in the format *Type(Name)*. Each upper layer command returns a List. The first element of this list is a Reply code. The second is another List which contains any information associated with that Reply code. The "return values" specified for each function below refer to the content of the List returned with the Success Reply code, unless otherwise noted.

2.3.1 Connection Commands

Each of the commands in this section is used in initializing a session with a particular MediaServer. All commands are sent using the transmission format specified in Section 2.2.1. Each command in this section is prefaced with either a **U:**, **I:**, or **A:** to denote whether it is sent over an Uninitialized, Initialized, or Authenticating connection. An uninitialized connection is one which has just been opened; this is the first message. An initialized connection is equivalent to an active connection. An authenticating connection is either an initialized connection, one which has sent a single uninitialized connection command which now requires one or more messages to authenticate the user, or an enqueued connection waiting for access to the service.

U:Ping

Queries the MediaServer for current status information, including but not limited to the date, the time, the number of active connections, the number of connection requests in the queue, the maximum queue size, the highest Protocol Version ({Major, Minor}) supported, and a list of the ProtocolID's of all supported authentication protocols.

A:Authenticate *Integer(MajorVersion) Integer(MinorVersion) Integer(ProtocolID)* ...

This command sends information sufficient to authenticate a user, using the protocol described by the ProtocolID. ProtocolID=0 is defined to require two unencrypted Strings, one containing a username the other containing a password. At least one authentication protocol must be supported. Other protocols can be supported. Any protocol which requires multiple request-response messages must use this command for each request.

If the service is full, meaning that the number of connections has reached some limit imposed by the system administrator,[5] the server should immediately return this information without authenticating the user.

The client sends the highest Protocol Version that it can access. Before authenticating, the server examines the Version numbers. If the server cannot support a client with that Protocol Version, it sends MSbadVersion with the lowest Protocol Version that it can support. Using the SetVersion command, and the MSbadVersion response, the client and the server can negotiate which Protocol Version to use for this session. The server may give up on the negotiation at any time by sending the MSnoVersion response.

A:SetVersion *Integer(MajorVersion) Integer(MinorVersion)*

This command is used in negotiating which protocol version the client and server will use. It is more fully described in the Authenticate command description.

U:Enqueue *List{Integer(MajorVersion) Integer(MinorVersion) Integer(ProtocolID) ...} List(Queuing Information)*

For servers which support connection queuing, this command is used to initiate a connection and inform the server that the client wishes to be

[5]Typically, this limit will be a function of the number of simultaneous users of the database allowed by the license held by the administrator. The administrator may also want to limit the number of users within some category. For example, a university might choose to ration the number of connections from outside the university in order to give students and faculty higher access priority.

placed on the connection queue if the service is full. The process by which a server maintains the queue is unspecified in the protocol.

For servers which do not support queuing, this command must still be supported. It should act like Authenticate, except that if the service is full it should return MSnoQueuing instead of MSserviceFull.

If the queue is full, the server should immediately return this information without authenticating the user.

A:Dequeue *Integer(ProtocolID)* ...

If authentication is successful, remove the user from the connection Queue. Note that this command is not specified for use on an initialized or uninitialized connection; that decision is left to the implementation of the queuing protocol.

I:Disconnect

This command is used to gracefully shut down a connection to a MediaServer. When this command returns, the server is prepared for the connection to be closed.

I:DisconnectConfirm

This command is used to acknowledge the ShutdownNow reply. After this command is sent, the connection is inactive and should be closed.

I:`disconnectRetriever` *String(Retriever host name)*

This command is used to gracefully shut down a connection to a Retriever. It requests that the connection to the specified Retriever be shut down. If there is no such Retriever active, this command has no effect. When the command returns, the Retriever is prepared for the connection to be closed. This command has the side effect of aborting any media transmission being performed over the connection to the specified retriever.

I:`mediaSend` *ObjectID(MediaElement)*

Requests that the Media Object referenced be sent to the Client. Returns the name of a Retriever host and a port on which to contact that Retriever. The invoking function is then responsible for setting up a connection to the host on the port or using an existing such connection.

2.3.2 Database Access Commands

With the exception of the commands for deleting objects, these commands parallel the commands in the Virtual Database[5, 6].

GetAllClasses

Returns a List of the names of all the classes/tables in the database.

GetBaseClasses

Returns a List of all the classes in the database which have no superclasses.

GetSubClasses *String(class name)*

Returns a List of the names of all the subclasses of the specified class.

GetSuperClasses *String(class name)*

Returns a List of the names of all the superclasses of the specified class.

GetAttributes *String(class name)*

Returns a List of Lists, each subList is a pair of Strings, the first of which is the attribute type, the second of which is the attribute name.

GetMethods *String(class name)*

Returns a List of Lists, each subList contains a method name, its return type and a List of the types for the arguments.

ExecuteStr *String(query)*

Executes the given query string. Returns a List containing a marshalled Cursor Object, which includes a List containing the types in each record and an Integer which contains the number of records in the cursor.

CursorOpenAt *ObjectID(Cursor ID) Integer(cursor position)*

A Cursor returned from a query execution contains a count of the number of records in the Cursor. It is the client's responsibility to do bounds checking on the CursorOpenAt() command. The first record in a Cursor has index 1, the last has index equal to the number of records in the cursor.

ObjectGetAttribute *ObjectID(Object ID) String(attribute name)*

Returns the value of the specified attribute for the specified Object.

ObjectCall *ObjectID(Object ID) String(method name)*

Calls the specified method on the specified Object and returns the result.

KillObjects

This "command" exists to send the Kill List to the Server. It has no other effect.

Object Removal

Objects which are deleted from the client should always have their ObjectIDs placed on the Kill List. This list is sent with every command. If an invalid ObjectID is found on the Kill List, it is ignored.

2.3.3 Media Request Commands

MediaAccessibility *ObjectID(mediaElement)*

Checks the Accessibility of the referenced object. If the object is not a Media Object, an error is returned. The Accessibility of the object is returned through error messages if it is unavailable, or through Success if it is available.

MediaSend *ObjectID(mediaElement)*

This command is executed on the client by sending the Connection command `mediaSend`. Given the resulting host name and port number, this command confirms that a connection to that host/port pair exists. If it does not, the command initiates one.

The command returns a handle to whatever type of data connection is being used for the Media Transport Connection. When it returns, the Media can be read off this connection.

MediaAbort

This command aborts the current Media transfer. If there is no transfer currently active, this command has no effect.

ListManipulations

Returns a List of all supported manipulations and their required arguments. Examples of the type of manipulations that a Retriever might support are running a particular compression algorithm on an element, cropping an Image, or extracting a particular frame from a Video, so that only the resulting Image need be shipped.

The specification of this command is purposely not complete.

MediaManipulate *ObjectID(mediaElement) List(Manipulation Data)*

Performs the specified manipulation on a copy of the specified object and returns a reference to the resulting new media Object marshalled in a List. The Retriever resources used to store the temporary version of the image are released when the server receives a reference to the object on the Kill List.

The specification of this command is purposely not complete.

2.4 Replies

The replies of the system are broken into three categories: those which are always possible return values, those which occur only during session initialization, and those which can occur during an active session.

2.4.1 General Replies

Success *List(Reply)*

This reply indicates that the Command sent was successful. If any return values were generated by the Command, they are available in the Reply List.

NetworkError

Indicates an error on the Network. This is a fatal error.

ServerError

Indicates some error on the Server. This is a fatal error.

ShutdownNow

Indicates that the MediaServer is shutting down. The only valid Command that can be sent at this point is a DisconnectConfirm.

DBfatal

Indicates an fatal error in accessing the database. Like ShutdownNow, the only valid Command that can be sent at this point is a DisconnectConfirm.

IllegalMessage

Sent when the previous message could not be parsed successfully or when the Command code is unknown. The erroneous message is ignored.

2.4.2 Session Initialization Replies

MSaccessDenied

The user could not be authenticated. Connection is Terminated.

DBinitError

The Database could not be opened. This is a fatal error.

MSserviceFull

The number of active connections is greater than or equal to the maximum number allowed. Depending on the queuing protocol supported, the server may disconnect or wait for an Enqueue request after this Reply is sent.

MSnoQueuing

This is the alternative to MSserviceFull which is sent in response to the Enqueue Command if the server does not support queuing and the service is full. Connection is Terminated.

MSqueueFull

This is sent in response to an Enqueue command when the connection queue maintained by the server is full. Connection is Terminated.

MSunsupportedAuthProtocol

The server does not support the authentication protocol the Client is attempting to use. Connection is Terminated.

MSauthResponse

This is the Reply message sent as an intermediate response for a multi-message authentication protocol.

MSbadVersion *Integer(MajorVersion) Integer(MinorVersion)*

This response is used by the server when negotiating the appropriate protocol version to use for the current client session.

MSnoVersion

This command is used by the server to discontinue protocol version negotiation. Connection is Terminated.

2.4.3 Active Session Replies

DBinvalidClassName

The database does not contain the requested Class.

DBnullQuery

The query string sent was empty.

DBinvalidQuery

The query string sent was not a valid query.

DBaccessDenied *(DBlocked, object locked, insufficient permission)*

The database denied access on the last access Command for the reason stated.

DBcursorOutOfBounds

CursorOpenAt was sent with an index out of bounds for the referenced Cursor.

DBinvalidObjectRef

The ObjectID sent was invalid.

RobjectNotMedia

The ObjectID sent does not reference a Media Object.

DBinvalidAttributeName

The object does not contain the requested Attribute.

DBinvalidMethodName

The object does not contain the requested Method.

DBmethodDispatch *(bad argument list, unknown)*

The database encountered an error while dispatching a method, for the stated reason.

DBunknown

Indicates an unknown non-fatal error in accessing the database.

RmediaUnavailable *(file locked, file not found, retriever not responding)*

The requested Media Object is unavailable for the listed reason.

RmanipulationNotSupported

The manipulation requested is not supported.

RinvalidManipulation

The manipulation requested is not valid on the Media type sent.

RmemoryError

The Retriever has insufficient memory to perform the requested manipulation. Delete temporary objects and try again.

2.5 Protocol Extension/Versioning

Since the Protocol specified herein is designed to be extensible, I also specify a standard for assigning version numbers to the Protocol as it evolves. Each version has two numbers, a major version number and a minor version number. Thus, each version is assigned a number major.minor. Any new version of the protocol which introduces a major semantic addition to the protocol, which alters the semantics of existing functions, or which removes function(s) is assigned a version which has the major version number incremented from its most recent value and a minor version number of zero.

Any other new version is assigned a version which has the minor version number incremented from its most recent value; the major version number remains unchanged. Servers supporting a particular major version should support *all* the minor versions of that major version series. It is less likely that any given server will support more than one major version of the protocol.

The original Image Server protocol [7], of which there are no implementations, is version 0.0. The version of the MediaServer protocol specified in this document is version 1.1[6] Whenever the MediaServer is modified to support other media types, the minor version should be incremented to 1.2. Finally, when write-capability is added to the Protocol, the Major version number should be incremented to 2.0.

[6]Version 1.0 is the first cut of the version 1.1 protocol. It seemed appropriate to only increment the minor version number for this specification, since again, there is no implementation of 1.0.

Chapter 3

Server Implementation

This chapter details the prototype implementation of a server conforming to the MediaServer protocol. The prototype was implemented in C++ using the Sun version 3.0.1 compiler, the Beta 2 Release of the AM2 utility, network, and database class libraries and Rogue Wave Software's Tools.h++ Class Library, Version 6. The prototype was built and tested on a Sun SparcServer10 and a Sun SparcIPC workstation, both running the Unix 4.3BSD operating system. The underlying database used for testing purposes was UniSQL's object-relational database product, the UniSQL/X Database Management System. The implementation described in this chapter is fundamentally dependent on the listed class libraries, and on the Unix operating system, especially with regard to networking issues.

This chapter is structured as follows. Section 1 describes the architecture of the server system, emphasizing the details of the architecture not specified in the protocol model. Section 2 describes how the prototype maps onto the protocol, specifying the representations used for the abstract List and Value types, and the underlying communication protocols. It also describes the elements of the protocol which are not implemented, or in some places incorrectly implemented. Section 3 details the operation of the components of the system. The operation of the client is held over to Chapter 4.

For the remainder of this chapter, the prototype implementation will be referred to as "*MediaServer*" and the protocol itself will be referred to as "the protocol."

3.1 Architecture

The protocol specifies a system with six logical elements: the physical database, the virtual database, the MERIS, the Media Repository (comprised of one or more Repository components), the Retriever, and of course the client. Our implementation adds two more components: the Connect Daemon and the Retriever Daemon. See Figure 3-1. The model in *MediaServer* specifies that each client connected to the server has its own dedicated MERIS. Likewise, each client connected to a particular Repository component is connected to a dedicated Retriever.

The Connect Daemon is a single process that waits for connections initiated by clients. This process performs all of the setup specified by the protocol, including all session-level user authentication, and any queuing required to enforce active connection limitations. When a user has been successfully authenticated and gained an active connection, the Connect Daemon starts up a MERIS process and passes control of the client connection to it. The MediaServer implementation of a MERIS handles all other commands of the protocol sent the client.

The Connect Daemon is responsible for maintaining a queue of pending clients when the service is full and for logging access information. It is also responsible for keeping track of all information necessary to effect an orderly shutdown of the system, as well as "clean" crash recovery.

Each Repository component runs a single Retriever Daemon. This process receives requests from MERIS's much in the same way that the Connect Daemon receives requests from clients. The Retriever Daemon is responsible for confirming that the connecting process is in fact a MERIS from the system. Once it has done so, it creates a Retriever process and passes control of the MERIS connection to it. Each time a Retriever is created (or destroyed), the Retriever Daemon must inform the Connect Daemon. The Retriever Daemon is responsible for maintaining a log of active Retrievers to assist the Connect Daemon in shutdown and crash recovery procedures. The Connect Daemon can instruct the Retriever Daemon to terminate any currently active Retriever, as well as instructing it to shut itself down.

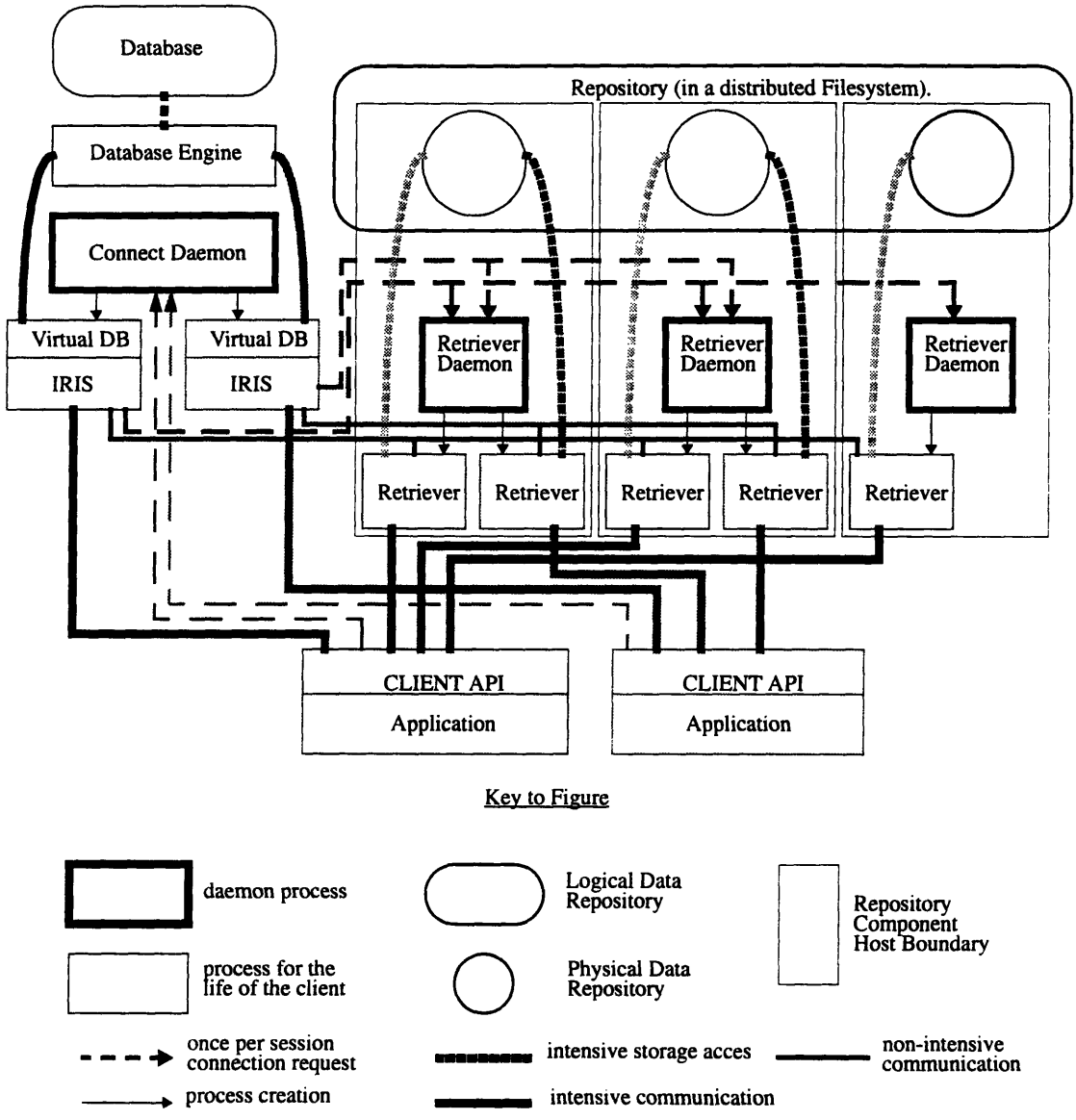

Figure 3-1: The Prototype *MediaServer* Architecture

3.2 Mapping to the protocol

The protocol specifically does not specify the implementation details for the communication layer. This section details the implementations used in the *MediaServer* for the abstract List and Value types, as well as the underlying communication layer and the required *Encode* and *Decode* functions. Section 3.2.1 describes the List and Value types. Section 3.2.2 discusses the underlying communication layer and some limitations it introduced. Section 3.2.3 describes the portions of the protocol that either were not implemented or are in fact incorrectly implemented in *MediaServer*.

3.2.1 The *List* and *Value* types

MediaServer is implemented on top of a layer of utility classes written for AM2. The abstract List and Value types map directly onto two of these, UTlist and UTvalue. Obviously, the protocol specification was heavily influenced by the availability of these classes.

The UTlist class provides exactly what the protocol specifies the abstract List type must provide. It is an ordered sequence of UTvalues. Methods to count the number of objects in the list and to access the UTvalue in a specific position are provided.

The UTvalue class fulfills almost all of the requirements of the abstract Value type in the protocol. It can directly contain all the required types except ObjectID and Binary. ObjectID is currently defined to be an integer (specified to be a `long int` as defined by the ANSI C specification). Since the only intended use of the Binary type is for encrypted authentication protocols, and since the focus of this thesis is not the security of the server, it was decided that this deficiency was acceptable. Adding a Binary type to UTvalue *is* possible and should not be particularly difficult, which further justifies this decision.

3.2.2 The Communication Layer

The communication classes for the MediaServer are built on top of the AM2 network classes, which are an implementation of Sun Microsystem's XDR standard for data encoding implemented on top of the TCP/IP reliable stream protocol [8]. The communication classes were designed by Masanori Kajiura in conjunction with the author. This layer is intended to encapsulate all necessary network communication functionality and process control. Thus, instead of executing a series of commands to setup a connection between the client and the server, all that is required is for the server to instantiate a communication layer server object with the appropriate well-known port and call the Listen method on the object. The client merely constructs a *MediaServer* connection object. The result on the client is an active connection; on the server a new connection is created for the client and the well-known port connection remains open.

Unfortunately, although the current communication classes are sufficient for a prototype *MediaServer*, they need to be redesigned slightly and re-implemented to provide a truly robust and extensible system. The common factor of all the problems is that the communication class design is based on an overly rigid model of *MediaServer*'s process evolution and communication patterns. Some specific limitations which have been discovered include:

- The communication layer was designed and implemented to conform to an architecture which predates the protocol specification. As a result, the communication layer imposes a certain level of incompatibility with the protocol. Most notably, most of the connection commands and responses of the protocol are implemented in the communication layer, with a general "high-level message" and "high-level error (response)" command in the communication layer allowing the remainder of the protocol to be implemented in the controlling application.

  The result is that connection commands and the mediaSend command are sent as a List structured as {`command, arguments`}, while all others are sent as {`HighLvlMessage, {command, arguments, Kill List}`}. In addition to the

incompatibility this introduces with the message format of the protocol, it also means that access to all the connection commands is through the interface to the communication layer, which leads to the next limitation.

- Limitations on the types of arguments passed with connection commands, and which commands are supported. The most notable limitation this introduces is that connection queuing following the protocol model is not possible. A design for a queuing facility is described in Section 3.3.1, but this design should be taken with a grain of salt, as it is not backed by a working implementation. In contrast, the general "high-level" message passing method simply passes a UTlist, which allows the format of this message to be arbitrarily altered without requiring a change in the underlying message passing method.

- The abstraction layering of the network classes is poorly designed. In order to allow multiple simultaneous connections to a process (necessary for one possible method of queuing) or timeouts on a connection (in order to periodically perform cleanup operations, for example), the Berkeley Sockets command `select()` must be used. This command requires a list of file descriptors representing connections to poll. Unfortunately, getting access to the file descriptors of a stream requires blatantly breaking the abstraction barrier the stream classes provide. Furthermore, once `select()` returns, it is necessary to map the active file descriptor back to the appropriate stream object. Unfortunately, this abstraction deficiency is a problem in the underlying AM2 network classes, not just in the *MediaServer* communication layer.

  The current solution to this problem is to simply break the abstraction barrier. A better solution considered was to provide a correct abstraction in the communication layer, such that the *MediaServer* is decoupled as much as possible from the communication layer implementation. The downside of this scheme is that it results in a tight coupling between the communication layer and the network layer. However, this is far more appropriate than the current situation which couples the server application to the underlying network layer (through

the communication layer), since the communication and network layers require similar expertise to modify, which is significantly different than required for the *MediaServer* components. This solution was discarded because altering the communication layer is non-trivial and future implementations of the underlying AM2 network classes should fix this deficiency.

- An overly restrictive model of the life of the system. This is actually a symptom of the fundamental problem with the "network" layer, which is that this layer provides both network communication and process control. An example of this problem is that when the client sends an image request to the MERIS, the communication layer intervenes and before the MERIS even gets the request, the appropriate repository's Retriever Daemon is contacted and a Retriever started if necessary. One inefficiency this could cause would be if a user were unauthorized to receive any images, the client would still successfully connect with Retrievers.... Furthermore, if an Retriever Daemon does not respond, the client is given an error, and the MERIS has no knowledge of the failure. This is obviously incorrect, since the MERIS may have the capability to reactivate an Retriever Daemon, while the client certainly does not.

MERIS/Retriever communication

The protocol does not specify the communication protocol between the MERIS and the Retriever. *MediaServer* uses the same facility for this communication as is specified by the protocol for the client to MERIS connection. The number of messages is obviously very small. This implementation protocol is presented in Section 3.3.2.

3.2.3 *MediaServer* deficiencies

There are a couple of deficiencies in the prototype other than those introduced by the communication layer as noted above.

The error reporting in the prototype leaves much to be desired. There are two major causes for this. The first is that the AM2 Virtual Database implementation

does not implement error reporting. Thus, any database errors are reported by *MediaServer* as DBunknown. Most database access errors just result in Null responses. The second cause is that the error reporting facility in the communication layer only allows a single string to be sent in any reply other than the Success reply. Additionally, communication and network layer errors are not reported as error strings, but rather as integer error codes.

As noted in Section 3.2.1, UTvalue does not support the Binary type. This is not a problem for the prototype, since it supports no encrypted authentication protocols.

The Manipulate command is unsupported. It is recognized by the server, but generates RmanipulationNotSupported regardless of the content or OID specified.

In the prototype, only the Image media type is supported. The main reason for this is that the database/repository against which the prototype was tested only contains image data.

3.3 Component Process Descriptions

3.3.1 The Connect Daemon

The *Connect Daemon* must be available at all times that the *MediaServer* is accepting connections. It "listens" for connections initiated by clients to the "well-known port" (currently 2020) on the MediaServer Connection host (machine). The connect daemon is responsible for most session-level maintenance functions, including:

- Authentication of users.
- Controlling the number of active connections to the system.
- Starting a MERIS for each active client connection.
- Session-level logging.
- Process "garbage collection" to ensure that all system resources have been freed when a client disconnects, and to kill frozen MERIS's and Retrievers.

- Performing an orderly shutdown of the system at the request of a system administrator.
- Safe crash recovery to ensure that any surviving client connections are not interrupted unless necessary for system restart.[1]

User Authentication

User authentication is performed entirely by the connect daemon. The prototype only supports the bare minimum of authentication, relying on client's sending login names and passwords unencrypted. In the future, adding either public key encryption or Kerberos authentication capability to the service will be necessary (especially if the *MediaServer* ever supports write access). The connect daemon authenticates the user by looking up the user's name in an internal table generated from an external file and verifying that the user has sent the correct password.

A *MediaServer* administrator is responsible for creating a map of username's (and passwords, and possibly host addresses) to database "logins" which provide varying levels of access. When a user is authenticated, the daemon looks up the appropriate database login and password for the user, as well as any system level restrictions associated with this identity or associated with the source host address.[2]

If a user cannot be authenticated, then the client is sent the MSaccessDenied response and the daemon breaks the connection. "Anonymous" logins can be enabled by placing a user "anonymous" in the table. The Connect Daemon will accept any string for the password, which is placed in the log with "anonymous." Users should send their full email address for the password.[3] The server indicates that it accepts anonymous logins by returning TRUE with the access denied message.

---

[1] The last three functions — process reclamation, shutdown, and crash recovery — are all closely related. Due to time constraints, their design was never fully implemented, although I believe that the design described herein, if implemented, would fully support these maintenance functions.

[2] An example of such a restriction might be not allowing access to images, or not allowed access to any elements stored on a specific machine within the repository. In every case, it is more appropriate for the restrictions to be placed on the database user id used by the MERIS on behalf of the user, but since the system is designed to be used with stable legacy databases, adding such restrictions may be infeasible.

[3] This behavior is similar to the FTP anonymous login capability.

Maintaining Active Connection Limits

If the requesting client is authenticated and the number of current users is less than the (configurable) limit, the connect daemon spawns a MERIS as a child process and passes control of the client connection to it. If the service is full (the number of active connections equals the limit), then the potential user is added to a list of "eligible users" and the client is sent the MSserviceFull error with a Ping response. The client has a limited time (returned with the response) to request enqueuing. If the client fails to connect again to request enqueuing before this time is up, the user is removed from the eligible queue. If the Enqueue command is authenticated and received in time, the user is added to the connection request queue. The client is required to supply a port number and hostname with the Enqueue command. When the connection can become active (*i.e.* there is an open license), the Connect Daemon initiates contact with the client, creates a MERIS, and proceeds as before. If the client does not respond, it breaks the connection and continues down the queue. At any time while on the queue, the client can send an authenticated request to be removed from the queue. While waiting on the queue, clients can send only two commands: an authenticated Dequeue request, and Ping.

Since the Enqueue command requires authentication, it can be sent as a session initiation command. As a result, even an Enqueue command sent after the user is removed from the eligibility queue will result in the connection being queued (although the user is likely to be farther back in the queue). The only difference is that it results in an automatic addition to the queue. The Enqueue command returns a Ping response. It is important to note that just as the number of active connections can be limited, so too can the size of the queue. If the queue is full, the MSqueueFull error is sent with a Ping response.

Ping Requests

The Connect Daemon interprets any attempt to log into the *MediaServer* with username "PING" (all capital letters) as a ping request. When the Connect Daemon receives a ping request, it responds by returning the current date and time, the num-

ber of currently active sessions, the number of enqueued connection requests, the maximum queue size, and a list of the protocolID's of all supported authentication protocols (see Section 2.3.1). If the password sent with a Ping request is non-null, it is assumed to be a "username@hostname" identifier. If such an identifier is observed, the server should add to the response a Boolean which is TRUE if the user is on the queue, and FALSE otherwise. If the hostname is the MERIS's host and the username is a number, the ping is considered a notification of normal status from the MERIS running with process ID equal to the number sent.

Connection Logging

The connect daemon is also responsible for logging all attempts to connect to the system. This single logical log is stored in three separate physical logs. The first is a log of all currently active connections, containing the login, database identity, length of time spent waiting in the service queue, and process ID of the MERIS for each client. The correctness of this log is important because it is used in in crash recovery. Any time that the log needs to be modified, a copy of the new log is written to a known temporary logfile. The original logfile is then deleted. The temporary logfile is copied to the original filename. Finally the temporary logfile is removed. This results in an atomic update of the log, since there are only four possible states of the two logfiles, two of which are unique (only the original logfile or only the temporary logfile exist), the other two of which can be distinguished by the creation time of the logfiles (the older logfile should always be used as the correct log).

The second log lists the users currently on the queue, along with the time at which they first attempted connection. It is maintained using the same two-file logging method as the first log. This log is not necessary for correct operation of the system, since a queued client can send the Ping request with a username to test for continued presence on the queue. It is, however, necessary for maintaining accurate access records.[4]

The third log is a single physical append-only file, which contains a log of connec-

[4]The second log is not used in the current prototype, since queuing is not implemented in it.

tions. Each entry of the log is in one of four basic forms:

- login (host) connect_timestamp - disconnect_time (duration) (Q: time)
- login (host) connect_timestamp ACCESS_DENIED
- login (host) connect_timestamp SF (Q: time)
- login (host) connect_timestamp QF

There is an entry for every attempted connection to the Server. Every entry contains the identity of the user (including the hostname from which they connected), and when they connected (in the form of a timestamp without the year). The first format is used when a successful connection is made. It additionally contains the time at which the user disconnected, how long that user had an active connection, and finally the amount of time the user spent in the queue before gaining an active connection. The second format is used when a user fails to authenticate. The third is used when a user is enqueued, but never gains an active connection (either because he removed himself from the queue, ignored a connection from the Server, or was enqueued when the Server crashed). The fourth format is used when the user was unable to connect because the queue was full, or because the Server does not support queuing.

Process Cleanup

In addition to the connection log, the daemon keeps a log which contains a list of all active server processes. At the head of this log is a list of MERIS's believed to be active, and the process id of the Connect Daemon which started each. After this list, all of the active Retrievers are listed, in the form:

`MERISpid, RetHost, Retpid`

where the first entry is the process ID of the MERIS to which the Retriever is connected, the second is the name of the host on which the Retriever is running, and the third is the process id for the Retriever.

As the system runs, the Retriever daemons are responsible for informing the Connect Daemon of Retriever creations and terminations. The connect daemon keeps

track of the MERIS's, by adding them to the active log when they are created and removing them when they die. Since MERIS's are child processes of the Connect Daemon, the operating system sends a SIGCHLD system interrupt to the Connect Daemon when one of them terminates. The Connect Daemon registers an interrupt handler which the operating system calls upon generating a SIGCHLD. This handler performs the appropriate logging.

Periodically, the connect daemon confirms that each MERIS is running and responding, by "pinging" it (The connect daemon is actually guaranteed to learn of the termination of a MERIS, so the real purpose of this is to confirm that the MERIS has not hung.). This is achieved through the use of the Unix pipe facility. Each time a MERIS is created, the Connect Daemon creates two pipes, one for communication from the Connect Daemon to the MERIS (CD→M) and one for communication from the MERIS to the Connect Daemon (M→CD). A ping is sent by sending a short message over the CD→M pipe. The MERIS detects this and replies on the M→CD pipe. In order to ensure that a MERIS is not improperly terminated while processing a long database query, the Connect Daemon must give the MERIS some reasonable period of time to respond. This timeout is configurable, and should be tuned for each implementation based on the response time of the underlying database.

If a MERIS fails to respond to the ping in time, the connect daemon forcibly terminates it, then goes through the log and ensures that all Retrievers associated with the MERIS are terminated. This prevents failed MERIS processes from continuing to hold database licenses.

System Shutdown

An orderly shutdown is one in which all processes are terminated through advance notification instead of merely being terminated by a parent process or being left to hang by a peer process (the process on the other end of a connection). An orderly shutdown is performed when the Connect Daemon receives a SIGTERM signal.[5]. To

[5]A SIGTERM signal is the default signal sent to a process when "kill pid" is executed, where pid is the process id of the process to terminate. Both cases require a user with permissions at the same

perform a shutdown, the connect daemon first shuts down the well-known port. Then each MERIS is told to shutdown. Each MERIS then shuts down all associated Retrievers and disconnects the client. When all these actions are confirmed, the MERIS exits. Meanwhile, the connect daemon flushes the queue of pending connections, sending Shutdown messages to each. After a (short) specified time, the connect daemon forcibly terminates all active processes. Finally, the daemon confirms that all logs are cleared, except the connection history log. It then connects to each Retriever Daemon in turn and instructs it to terminate. The daemon then exits.

Crash Recovery/System Startup

MediaServer is implemented so that active sessions continue for as long as possible in the case of a partial system failure. If the Connect Daemon terminates for any reason other than a system shutdown, all currently active sessions can continue normally. If the Connect Daemon does crash, it must be restarted, either manually by a system administrator, or automatically through the Unix clock daemon "cron." When the Connect Daemon is restarted, it performs the following actions, in order:

1. Upon restart, the Connect Daemon verifies that it is the only such process running. If another exists, the Daemon verifies that it is active by sending it a Ping. If it responds, the new Daemon exits. If not, the Daemon forcibly terminates the old Connect Daemon process (note that this means that the Daemon must be started by either the same user identity as started the other Daemon, or by root).

2. The new Connect Daemon reads the log of users who had active sessions when the old Connect Daemon crashed. It examines the list of active processes on the system and removes from the log any MERIS's which have terminated.

3. The Connect Daemon now instructs each Retriever Daemon to terminate any "orphans"—active Retrievers from sessions with now-dead MERIS's. Once this

level or higher than the Connect Daemon was started with. SIGTERM is also sent to all processes running on a host when the host is shut down. SIGTERM can be caught by a process, allowing it to perform some action(s) before exiting.

is done, each Retriever Daemon is queried for a list of all still-active Retrievers, which the Connect Daemon uses to finish synchronizing the log with reality.

4. The Daemon must now verify that the database engine is running, by connecting to it and performing a GetAllClasses command. If the database is not running, the Connect Daemon executes a partial system shutdown, terminating all active MERIS and Retriever sessions. This is done by sending a SIGTERM signal to each MERIS, waiting for the normal timeout period, then repeating steps 2 and 3, with the change that any MERIS still active is forcibly terminated. Once this is complete, the Daemon restarts the database. If it cannot restart the database it appends an error message to the connection history log and exits.

5. Lastly, the Connect Daemon opens the well-known port and begins accepting new connection requests.

As long as the active log contains a MERIS which is not a direct child of the current Connect Daemon, the Daemon must behave slightly differently. Each time that it performs a "ping" of all the MERIS's, it also must also confirm that any non-descendent MERIS's have actually sent a normal Ping to the Daemon since the last round of pings.

The only modification of the above for a cold system start (a startup performed after an orderly shutdown) is that all Retriever Daemons should be activated before the Connect Daemon. This can be achieved by giving all Retriever Daemons cron check times slightly before the Connect Daemon's.

3.3.2 The Retriever and Retriever Daemon

The Retriever Daemon has very limited functionality. It will spawn a new Retriever at the request of a MERIS, and it will terminate its Retriever children processes on behalf of the Connect Daemon. It can also respond to a Ping from any location, returning the current number of active Retrievers. Finally, it can send a list of Retriever process IDs at the request of the Connect Daemon.

The Retriever Daemon maintains a log of currently active Retrievers and an append-only log of unauthenticated access requests. Before spawning a Retriever, the daemon will authenticate the MERIS request by means of a username and password, set by the system administrator. If an unauthenticated access request is made (other than a Ping), the Daemon logs it. When the MERIS is authenticated, the daemon assigns a port for the client to interact with the Retriever, creates the child Retriever process and passes control of the MERIS connection to the child. The MERIS can then initiate commands to the Retriever. When the MERIS sends the first sendMedia command to the Retriever (which sends the name of a file to send to the client), the Retriever responds with the size of the media element being retrieved. The Retriever the immediately begins sending the media.

The Retriever Daemon monitors Retrievers using the same pipe method used by the Connect Daemon to monitor the MERIS's. This method still works during large media element transport, for the same reason that Media transport can be aborted. After the Retriever writes a portion of data to the client connection, it calls `select()`. It is waiting for any of three actions: the Retriever Daemon's pipe is ready to read, the MERIS connection is ready to read, and the client connection is ready for more data. The first results in a Ping/Response interaction between the Retriever and its daemon, the second results in the media transport being aborted, and the third causes the Retriever to write the next block of the data being transferred.

Media element size and availability can be requested from the Retriever. This information is determined from the information that the Retrievers have about the files containing the elements. In the future, the Retriever will be capable of performing limited element manipulation (*i.e.* it will support the MediaManipulate command), such as compression, image cropping and dithering, *etc.* Each time a MediaManipulate is executed successfully, the Retriever stores the result in a temporary file and returns the filename to the MERIS, which is responsible for creating a temporary Media Object to return to the client. This object creation will be described in the next section in the discussion of object system maintained by the MERIS.

Retriever Communication protocol

There are only three commands supported by the prototype implementation of the Retriever:

MediaAccessibility filename

> Checks the accessibility of the filename. Returns a code indicating whether the file is available, not found, or locked.

MediaSend filename

> If the file is available, the Retriever returns its size in bytes. It then immediately begins to write the image to the open connection it has to the client.

MediaAbort

> This command aborts the current Media transfer. If there is no transfer currently active, this command has no effect.

3.3.3 The MERIS

The first task the MERIS has is to connect to the database, using the username and password the Connect Daemon looked up in the user table. If this is unsuccessful, the MERIS returns a DatabaseError message and shuts down. Otherwise, it enters its main loop, during which it waits for one of two activities.

First of all, it can receive a ping request over the pipe it has from the Connect Daemon. In this case, it writes a reply to the pipe.[6] If a MERIS has not received such a ping request in a configurable amount of time, it attempts to write to the pipe

[6] Actually, this is a three step process. First it must register a handler for the SIGPIPE interrupt, which is sent if an attempt is made to write to a broken pipe. Then it writes to the pipe. Finally it unregisters the handler. The handler is necessary because on some Unix systems, the SIGPIPE interrupt is also used to signal an attempt to write to a broken stream connection. Performing the above registration/unregistration ensures that the correct interrupt handler is called in the correct situation.

to the Daemon. If it is successful, the MERIS resets its wait clock. If it is unavailable (signified by a broken pipe), the operation of the MERIS changes slightly. Instead of waiting for a ping over the Connect Daemon's pipe, the MERIS now periodically sends a Ping to the Connect Daemon, including its own process ID as the username.

The second activity the MERIS can encounter is obviously a request from the client. All database access and media request functions specified in the protocol are implemented to perform their functions exactly as specified in the protocol. The only interesting aspect of the MERIS is the object handling system it uses to export references to the Virtual Database objects it contains. This system is described in the next section.

The Object System

The "object system" in *MediaServer* supports the export of objects from the server to the client as per the protocol specification. The abstract Object type may be present in a Value contained in a List. In the case of UTvalues, the Object type is called UThandle. Each UThandle in a UTlist can be thought of as a pointer to a Virtual Database Model object. If just this pointer were shipped to the client, it would have no meaning, since the client cannot use that pointer to reach the object. Instead, each UThandle must be replaced with something which can be unmarshalled on the client into a client object which has some kind of reference to the actual object on the server. Thus, any UTlist which might contain a UThandle must undergo a marshalling procedure which marshals any UThandles the UTlist does contain. Figure 3-2 provides an example of a UTlist which needs to be marshalled for sending to the client.

To maintain the mapping between server objects and client objects, the server has a utility class called MSserverObjects, which inherits from a general purpose utility class called MSremoteObjects. These classes are responsible for keeping track of all the active objects in the system and assigning a unique identifier to each. The process is fairly straightforward. The MERIS has a single MSserverObjects object. This object contains a counter for generating object identifiers (OID's), which is initialized

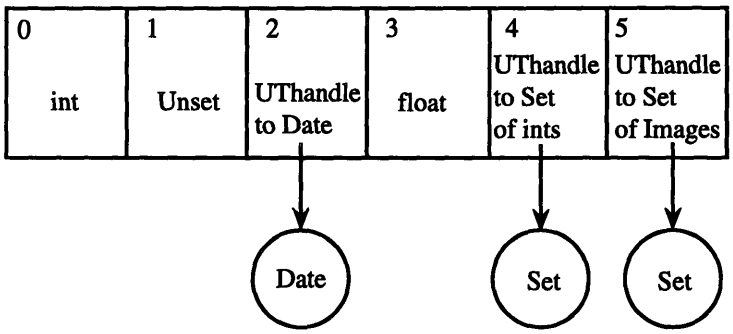

Figure 3-2: A UTlist in need of marshalling. The Unset Value in position 1, the Date in position 2, and the Sets in positions 4 and 5 all require marshalling before this List can be sent to a client.

to 1. It also contains two *Hash Dictionaries* for keeping track of active objects. A Hash Dictionary is a Hash table which allows for efficient storage of Value's indexed by Key's. The Value stored for each object in a Dictionary is a UTlist containing three elements: the object's OID, an integer code representing the object's type, and a UThandle to the object. The first Dictionary (`oidDict`) stores this Value using the OID as the Key, the second (`hanDict`) uses the UThandle as the index.

Marshalling always takes a single UTlist and returns a UTlist with three elements: the marshalled UTlist, a list of indices into the first UTlist indicating which elements are marshalled handles, and a list of indices into the first UTlist indicating which elements are UTlists containing marshalled handles in them somewhere. The procedure for marshalling a list `input` proceeds in the following steps:

1. First the UTlist is scanned. If there are no UThandles in the list (a deep search of all UTlist elements on the UTlist is made), and this is a top-level list then the UTlist {`input`, {}, {}} and the Eboolean eFALSE are returned. If this is not a top-level list, then the list is returned unmodified and eTRUE is returned (the reason for this distinction is explained in step 5). Otherwise, go to step 2.

2. Three temporary UTlists are formed: `tmpList`, `hanList`, and `listList`, corresponding to the three lists which comprise the final list sent to the client.

3. Each element of `input` is examined in order. If an element is neither a UThandle nor a UTlist, then it is simply appended to `tmpList`.

4. Each UThandle of the list is marshalled into a UTlist in a separate function (described below). The resulting UTlist is added to `tmpList`, and the position of the UThandle within `input` (and `tmpList`) is appended to `hanList`.

5. Each UTlist of the list is marshalled by recursively calling this list marshalling procedure, with the optimization that if the list contains no handles anywhere, then the list is returned unmodified. The marshalling function returns eFALSE when it marshals a list with no UThandles. If this happens, then the list is simply added to `tmpList`.

6. If the marshalling function returns eTRUE, this indicates that the list returned is a list which must be unmarshalled by the client. Thus, the returned list is added to `tmpList`, and its position in the UTlist is appended to `listList`.

7. Finally, if the list contains a UThandle anywhere, return eTRUE.

Each UThandle is marshalled in a separate function. Each UThandle is fully marshalled at most once in a given response. This prevents looping on Sets which may contain circular references. This is achieved by the MSserverObjects maintaining a list of all of the OID's which have already been marshalled into this reply (the `OIDlist`). Once the reply is sent, the `OIDlist` is cleared. A UThandle is marshalled into a UTlist containing four elements: an OID, the type of the object (coded in a C++ enumeration), a UTlist of attribute data, and a flag which is eTRUE if this handle appears somewhere else in this reply (in which case the UTlist of attribute data is empty).

The UThandle marshalling function performs the following steps:

1. UThandle is looked up in `hanDict`. If a match is found, use the OID in the Value, and look this OID up in the `OIDlist`. If it is found, no further marshalling is necessary, as the UTlist {`OID, Type, {}, eTRUE`} can be sent (where Type is also extracted from the Value). If it is not found, skip to step 3.

2. If no match is found in `hanDict`, a new OID is generated by using the current value of the OID counter, and incrementing it.

3. The class to which the object belongs is determined. The object must belong to one of Virtual Database "base type" classes: DTdate, DTmonetary, DTtime, DTtimestamp, DTmedia, DTset, or DTobject. The first four result in all of the object's attribute data being sent to the client. A DTmedia must be further resolved into a DTimage, DTvideo, DTaudio, or DTfullText.[7] The attribute data sent with a DTmedia are the OID of the predecessor object and the OID of the base object for the given object.[8] A DTset's attribute data is a UTlist of elements in the set. This list must be marshalled before sending. A DTobject's attribute data is simply the name of the object class.

4. Finally, the handle is formed into its marshalled representation of {`OID, Type, UTlist of attribute data, eFALSE`}. The Value {`OID, type, UThandle`} is inserted into both Dictionaries, using the OID or the UThandle as the key, as appropriate.

The result of performing the above function on the example UTlist is shown in Figure 3-3.

Whenever an object is referenced in a command, such as ObjectCall, the OID passed as a parameter is looked up in `oidDict`. If no match is found, the MERIS returns an error. Otherwise the UThandle is extracted from the Value, the pointer to the actual object is extracted from the UThandle, and the desired action is performed on the object.

Whenever any command is received, the Kill List is examined. Any OID's found on the List are looked up in `oidDict`. If found, the UThandle is extracted from the Value, and the Value is removed from both Dictionaries. The method DestroyAndNull is called on the UThandle, destroying the underlying object and invalidating any remaining references to it. If the OID is not found, it is checked against the Cursor

[7]This prototype only handles DTimage.

[8]These are attributes which exist to support the Manipulate command, which is unimplemented in the current *MediaServer*. The predecessor object is the object which was manipulated by the server to get this object; the base object is the object native to the server which is the original object in the chain of manipulations leading to the current object. In this implementation, these two OIDs are always the NullOID (OID=0).

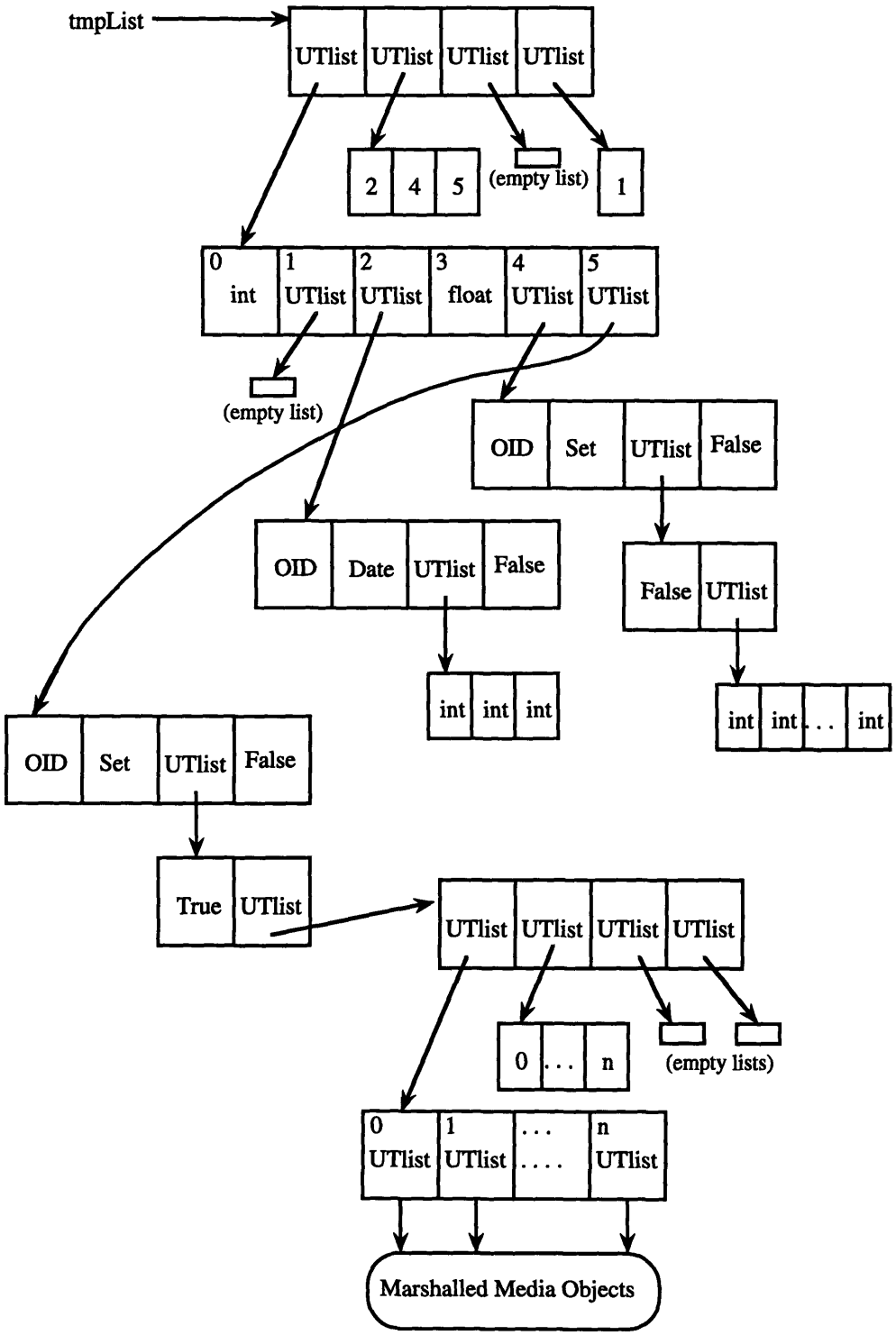

Figure 3-3: The List from Figure 3-2 after marshalling. The most complicated marshalling is of the Set of Images originally in List position 5. The UTlist of Images in the Set is exposed when it is marshalled. This list needs is marshalled in turn, requiring the marshalling of a number of Image Objects into Media object references.

list before being discarded.

Because of a limitation of the DTcursor class which prohibits it from being accessed through a UThandle, the Server must treat Cursors separately. The way this is done in the current implementation is to assign OID's from the normal progression of object OID's to Cursor objects, but not to register them into the Dictionaries. Instead, they are added to a special linked list of structures containing only an OID and a pointer to a DTcursor. When an OID on the Kill List is not found in `oidDict`, this list is searched for the OID. If found, the corresponding DTcursor is deleted and the structure removed from the linked list.

Because of a deficiency in the AM2 network layer, UTlist's containing Unset values are truncated at the first Unset value. As a result, the current implementation of the *MediaServer* marshalling code has the additional responsibility of marshalling Unset values. This is done by introducing a fourth UTlist in a marshalled UTlist which contains the positions of each list element which should be marshalled into an Unset value by the client. The Unset value of the `input` list is replaced by an empty UTlist on the `tmpList` list.

Chapter 4

Client API Implementation

This chapter describes both the interface and implementation of a library of C++ classes which can be used to access the *MediaServer* prototype implementation described in Chapter 3. This implementation suffers from the deficiencies that the prototype server exports to its clients, but introduces no new limitations. Section 4.1 describes the interface of the API while Section 4.2 discusses any implementation issues solved in the client library.

4.1 The MSclient class library

The client API provided as part of the prototype implementation of the MediaServer protocol is provided in a class library. This library contains eleven classes which provide full access to a Media Server. This section contains a summary of the functionality provided by each class and how it is used. A full description of the class library can be found in Appendix A.

MSmediaserver

MSmediaserver is the main point of connection to a *MediaServer*. Almost all objects which are created as a result of a connection to a particular server have a pointer to the object of this class which represents the connection to that server.

A MSmediaserver object supports methods which open a connection to a *Medi-*

aServer, close such a connection, retrieve error information from the last command invoked on the connection, retrieve the schema of the underlying database, and execute queries.

MSquery

An object of this class is totally independent of any MSmediaserver objects. An application can set a query string directly or build it up in lists representing each of the elements of a query. A query can be sent directly to the Execute method of MSmediaserver for execution. A single query can be executed on any number of MSmediaservers.

MScursor

This class is used to access the results of a query that has been executed on the *MediaServer*. It provides access to the results without requiring the entire result to be transmitted from the server to the client upon query execution. This delayed binding of the query result to client objects eliminates wasteful overhead upon query execution at the expense of possible small delays upon retrieval of each result.

All of the methods of this class except one operate locally on the data sent when the Cursor is first sent from the server. The Open(i) method returns the ith record in the Cursor.

MSobjectRO

This class represents any object in the database which cannot be represented using any other Virtual Database Base Type (integer, real, string, Boolean, Date, Monetary, Time, Timestamp, Set). It can be used to access the attributes and methods defined on the object.

MSdateRO

MSmonetaryRO

MStimeRO

MStimestampRO

MSsetRO

These five classes are fully instantiated on the client when shipped from the server. Thus, accessing them never results in any network traffic beyond the original transmission from the server. Each class represents the Virtual Database base type class of the same name.

MSsetRO represents a set of some type as a UTlist containing only elements of that type. Thus, instantiating an MSsetRO on the client can result in the instantiation of many objects.

4.2 Implementation Issues

The implementation of the client API is fairly straightforward. Only two issues had to be solved. The first was how to handle method invocation on MSobjectRO's. The second was how to implement the client side of the object system described in Section 3.3.3.

4.2.1 Method Invocation

While the client API was being implemented, a problem was discovered with MSobjectRO method invocation. There is no way to guarantee that a method defined on an object obeys the read-only semantics of the MediaServer protocol. This was a serious problem, since the security concerns of the *MediaServer* were subordinated to other concerns. There were really only two options available. The first was to not implement the ObjectCall command of the protocol until the protocol is augmented to allow read-write semantics. The second was to document the security hole and rely

on system administrators to set up their databases in such a way that the read-only semantics of the current MediaServer protocol are preserved.

Neither option is very appealing. It was decided that since this is a research prototype, it was more important to supply as much access to the *MediaServer* as possible. As a result, method invocations are supported, without restriction. Thus, if *MediaServer* is ever deployed in its current form, it should be altered to either not support ObjectCall, or to support a more secure authentication protocol.[1]

4.2.2 The Client Object System

An Object System on the client manages objects exported from a single server. Each server to which a client is connected has an independent network of objects. At the center of each network is an MSmediaserver object, which manages the connection to the server. MSmediaserver uses another object, MSclientObjects to manage the server objects instantiated on the client. Data classes are broken into two categories. Classes of the first category, called "local" objects, are MSdateRO, MStimeRO, MSmonetaryRO, MStimestampRO, and MSsetRO. Local objects can be fully instantiated on the client using the information supplied by the server. The second category of classes, which result in "remote" objects, is comprised of MSobjectRO, MSmediaRO (and its subclasses), and MScursor. These objects are not fully instantiated on the client, as they require actions by the server to access some (or all) of their data. Note that once the media element referenced by an MSmediaRO object has been retrieved from the server to the client, it becomes a local object.

The relationships among these objects are shown in Figure 4-1. In this figure, an arrow from an object A to another object B indicates that A has direct knowledge of the existence of B (*i.e.* A contains a pointer to B). The method GetObjectSys() supported on MSmediaserver gives objects access to the MSclientObjects object. The link from an MScursor or MSobjectRO to GetObjectSys() in MSmediaserver indicates

[1]Even this "fix" is not sufficient, since every command is not authenticated. Adding an authentication requirement to every method invocation would be sufficient, but would introduce significant overhead.

that these objects use the method.

Unmarshalling

An MSclientObjects object is used to maintain the mapping between client objects and server objects (represented by OID's). MSclientObjects inherits from the same general purpose utility class as the server, MSremoteObjects. The use of these classes on the client mirrors the use of MSserverObjects and MSremoteObjects on the server. The client has a single MSclientObjects object, which contains one Hash Dictionary, `oidDict`, for keeping track of active objects.

Unmarshalling a UTlist on the client is the inverse of the marshalling procedure on the server:

1. The UTlist is broken into its component Lists: `input`, `hanList`, `listList`, and `unsetList`. The output list is placed into `output`.

2. Each element of `input` is examined in order. If an element is *not* a UTlist, it is simply appended to `output`.

3. If a UTlist is encountered at location i of `input`, check to see if the integer i is a member of `hanList`, `listList`, or `unsetList`. If not, the UTlist is simply added to `output`.

4. If i is found in `hanList`, the UTlist is a marshalled UThandle. It is unmarshalled and the resulting UThandle is appended to `output`.

5. If i is found in `listList`, the UTlist is a marshalled UTlist containing at least one marshalled handle. Recursively call this unmarshalling procedure on the UTlist.

6. If i is found in `unsetList`, the UTlist is an empty list representing an Unset value. An Unset UTvalue is appended to `output`.

Unmarshalling a marshalled Object is performed in a separate function. Again, this function is the inverse of the server marshalling procedure. This function is

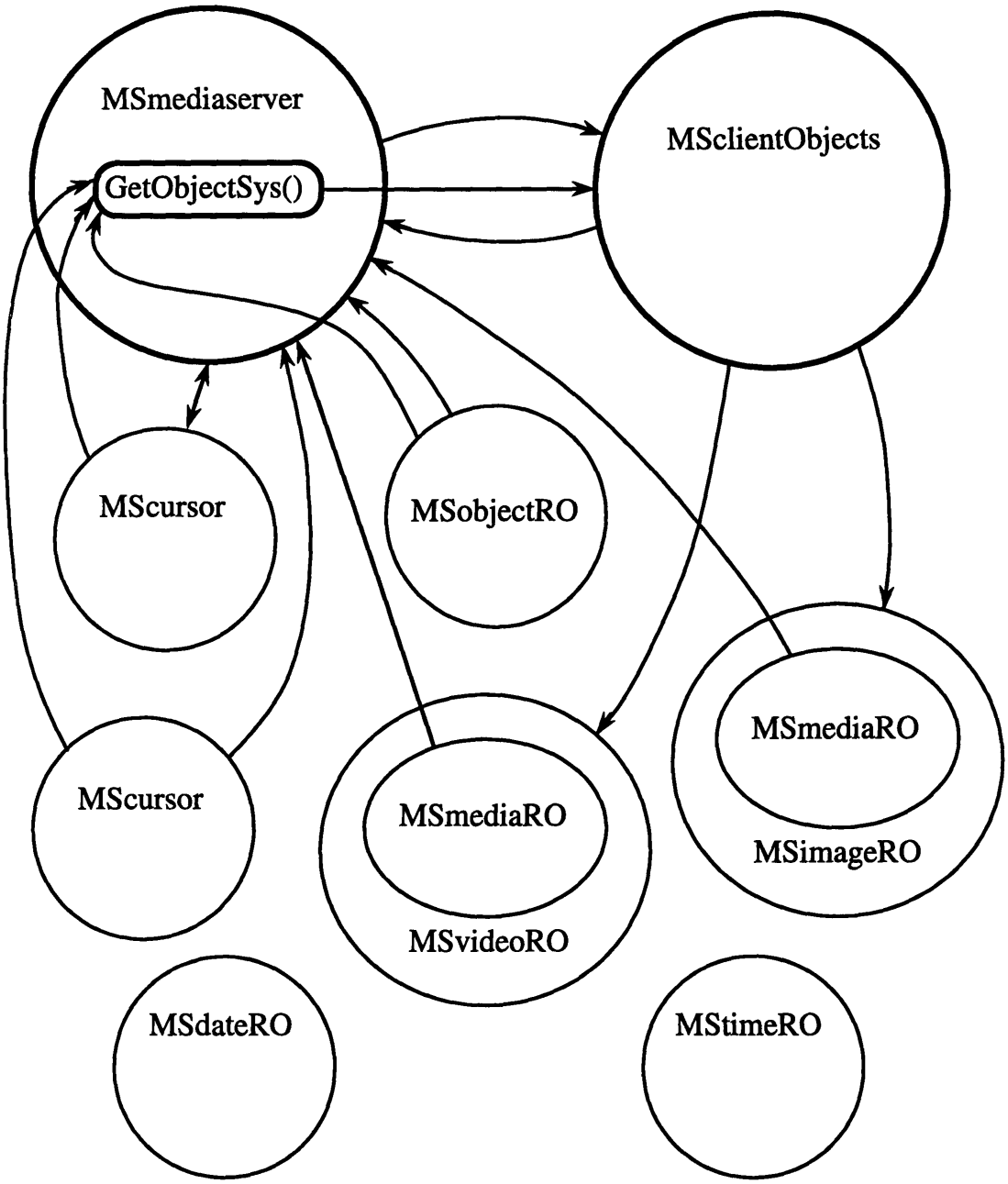

Figure 4-1: **Client Objects**: An arrow from Object A to Object B indicates that A has direct knowledge of B. There is only one instance per connected server of each of the MSmediaserver and MSclientObjects classes. GetObjectSys() is a method supplied by MSmediaserver which gives access to the associated MSclientObjects object.

slightly optimized for read-only semantics, since the `oidDict` is always checked for the OID of the object, and as a result the flag indicating whether this is a repeated object from the current message is never checked.

1. The OID of the object is extracted from the UTlist and is looked up in `oidDict`. If a match is found, the UThandle is extracted from the Value returned by the check and is returned.

2. If no match is found in `oidDict`, the object type and its data list are extracted from the input UTlist. A new object of this type is constructed using the data list and the OID.[2]

3. If the object type is Date, Monetary, Time, Timestamp, or Set, the OID is put on the Kill List maintained by the MSmediaserver object, since these objects exist independently of the server.

4. If the object type is Media, the type of media element is extracted from the data list, and the appropriate object is created.

5. If the object type is Object or Media, the Value {`OID, type, UThandle`} is added to `oidDict`, using the OID as the Key.

As on the server, Cursors on the client are handled slightly differently. When MSmediaserver's Execute method is invoked, the command ExecuteStr is performed by the server. This returns a marshalled Cursor, which Execute unmarshals without the use of MSclientObjects. The resulting MScursor object is not registered in the MSclientObjects's `oidDict`, but it is still destroyed using the MSmediaserver's KillOID() method, as described in Section 4.2.2.

Accessing Remote Objects

MSmediaserver has a method, DispatchMessage(), used by remote objects to execute commands on the server. When an object has a method invoked on it which requires

[2]Each object needs to know its OID, both for method invocation and deletion purposes.

action by the server, the object creates a message which is a UTlist containing the code of the command to execute and a UTlist of the arguments to that command, the first of which must be the object's OID. Thus, MSobjectRO's getAttribute(name) method executes the following code fragment:

```
UTlist message;
UTlist Args;
UTstring result;
Eboolean success;

Args << OID << name;
message << eObjectGetAttribute << Args;
success = mpServer->DispatchMessage(message, result);
```

DispatchMessage() returns eTRUE if the command was executed successfully. When it fails, `result` contains an error message. If it succeeds, `message` contains the returned UTlist. This list has not been unmarshalled, since some commands return UTlists which need not be unmarshalled.

Object Destruction

MSmediaserver provides a function, KillOID(), which takes an OID as an argument and frees the object referenced by that OID on the server. This is done by adding the OID to the Kill List (which is sent to the server every time MSmediaserver sends a command), and by calling MSclientObjects's deleteByOID(OID) method, which removes the object from `oidDict`. When any remote object is destroyed, KillOID is called to remove all references to that object in both the client's object system and the server's object system.

Chapter 5

Performance and Future Work

5.1 Performance Analysis

The design of *MediaServer* has been optimized to minimize the number of network connection creations and network messages required to achieve the design goals of the prototype implementation. To demonstrate, I will evaluate the performance of the *MediaServer* architecture in terms of the number of network connection creations, process creations, and network messages required to initiate a session, access the database, and ship media elements. I will compare *MediaServer* to other options for similar multimedia access. Performance *testing* was considered, but rejected because experience has shown that the performance of the system is dominated by media delivery and database actions. An argument can be made that performance analysis is equally unnecessary, but demonstrating that the system has no frivolous overhead is worthwhile.

5.1.1 *MediaServer*

Assuming an active client, initiating a session with *MediaServer* requires one process creation, one network connection creation, and two network messages. The client must create a connection to the Connect Daemon and send a message to it. The Connect Daemon then creates a MERIS, which sends a reply message back to the

client. More intensive authentication protocols can increase the number of messages, but the *system* only imposes these two.

Accessing the database through a MERIS requires a simple command-response message pair. Additional overhead imposed by the MERIS on a database transaction is the marshalling/unmarshalling required to export objects, and the delayed delivery of records in the cursor. This delayed delivery is seen as a performance improvement, since the only overhead imposed by it is an extra pair of network messages per record retrieved, and the benefit is that undesired records are not shipped. Furthermore, given this design, a client can be pipelined such that it simultaneously displays one record while retrieving the next.

The performance characteristics of media shipment vary depending on whether it is the first element to be retrieved for the client from the given Repository component. If it is, the overhead is two network connection creations, one process creation, and four messages. The client sends the sendMedia command to the MERIS, and the MERIS sends a command to a Retriever Daemon. The Daemon creates a new Retriever which sends back the size of the element being retrieved and the port number for the client to contact. The MERIS sends this information back to the client in the fourth message, and finally the client sets up a connection to the Retriever, which immediately begins sending the data.[1]

If the client requests an element which will come from a Retriever to which it is already connected, the overhead is four messages. The client sends the sendMedia command to the MERIS, which forwards the command to the appropriate Retriever. The Retriever responds with the size of the element, and begins to send the image. The MERIS forwards the size back to the client, along with indication of which Retriever is sending the element.

These overhead figures are summarized in Table 5.1.

[1]In the current implementation, the number of messages is actually eight, because setting up the client-Retriever connection is separated in the communication layer from the request for media transmission.

Action	Connection Creations	Process Creations	Network Messages
Initiate Session	1	1	2
Access Database	0	0	2
Retrieve Delayed Object Info	0	0	2
Ship Media (new Retriever)	2	1	4
Ship Media (old Retriever)	0	0	4

Table 5.1: The number of connection creations, process creations, and network messages required for the client to perform each type of *MediaServer* command.

5.1.2 Other Options

Retaining several features of the design of *MediaServer* is necessary to preserve its generality, but doing so limits alternative architectural options. First of all, *MediaServer* does not require that a client have file access to the server. This eliminates any option which involves the client simply mounting the Repository through a networked file system. Second, *MediaServer* allows an administrator to grant temporary access to the database through the use of virtual users, and to utilize any security protocol he wishes. Any option which requires the user to have direct access to the database restricts the administrator's security options. Finally, *MediaServer* does not even require that the connect host (the host running the Connect Daemon and the MERIS's) have direct file access to the Repository components, relying entirely on the Retrievers for accessing them. One major implication of this is that the Retriever is not required to store elements in files. Instead, the "filename" sent by the MERIS to the Retriever can be an index into whatever storage scheme the Retriever uses.

The above limitations eliminate the most obvious choices for architectures, including:

- The user connects directly to the database (using the Virtual Database Classes, of course), which returns filenames for media elements. The user is then responsible for extracting the elements he wants. This option is obviously less efficient anyway, due to its lack of integration. Specifically, *MediaServer* integrates database access with media retrieval. Doing so allows *MediaServer* to use a single connection (to a single Retriever) to retrieve many elements (resid-

ing natively on that Retriever's Repository component). This allows the cost of connection setup overhead to be amortized over all these retrievals. Furthermore, the centrally integrated nature of *MediaServer* allows security to be centralized. Since authentication is very expensive, this integration allows authentication to also be amortized over many accesses to the system, instead of each access having to be authenticated (an expensive proposition, at best).

- The user runs a client application which accesses the database directly. The database returns host names and filenames, and the client uses FTP to retrieve them (very much like the current World Wide Web). This option is not viable because it has virtually no security.

The only real option which achieves nearly all of the *MediaServer* capability is to eliminate the Retrievers. Instead, each Repository component runs an FTP daemon and a Repository Daemon. Whenever a MERIS needs to expose an element stored on a particular Repository component, it contacts the appropriate Repository Daemon. The Daemon creates a temporary file name for the element and exposes that file name to the FTP daemon. It then returns the temporary file name to the MERIS, which returns it to the client. The client then contacts the FTP daemon (or uses an existing connection to it) and retrieves the file. The Retriever Daemon eliminates the temporary file name after a limited amount of time. The performance for this system is identical to *MediaServer* except in media transfer.

In the case of a new Repository component being accessed, the alternative system requires three connection creations, some number of process creations, and eight network messages. The three connections are the one from the MERIS to the Repository Daemon, and the two connections specified by the FTP protocol. The network messages are the four explained above to get the temporary filename assigned, two to set up the data transfer connection for FTP, and the two (message-response) required to initiate the actual data transfer. The number of process creations is either zero, one, or two, depending on the implementation of the FTP daemon and the Repository Daemon. Each of these Daemons may create processes to handle each connection

Action	Connection Creations	Process Creations	Network Messages
Ship Media (new Repository component)	3	0	8
Ship Media (old Repository component)	0	0	6

Table 5.2: The number of connection creations, process creations, and network messages required for the client to perform media element retrieval on an FTP-based alternative to *MediaServer*. All other commands impose the same overhead as *MediaServer*.

they receive, or they can each handle them all. For this discussion, I will assume that both daemons handle all of their connections simultaneously.

Subsequent element requests require zero connection creations, zero process creations, and six network messages. The network messages required are the same as in an initial transfer request, without the two required to setup the data connection. It is possible for an FTP server to shutdown its data connection after each transfer, in which case subsequent transfers require one connection creation and eight network message. I assume that a reasonable implementation is used, which is capable of holding this connection open.

These overhead figures are summarized in Table 5.2. This alternative to *MediaServer* shows comparable performance overhead. When using existing connections, *MediaServer*'s four messages are slightly better than the alternative's six. Relative to the number of transmissions required to transfer a media element, this two-message difference is very small.

When accessing a Repository component for the first time, the alternative's zero process creations and 3 connection creations are more efficient than *MediaServer*'s 1 process creation and 2 connection creations. Again, given the time spent actually performing the transfer, these differences are negligible. Given the comparable performance characteristics, I believe that *MediaServer* is the architecture of choice, given its superior security support.

5.2 Future Work

I believe that the protocol and the prototype are an early step on a road of exploration into how to provide uniform views on widely diverse data types and storage schemes. In conclusion, I suggest the directions below for future work on the MediaServer protocol and its implementation.

5.2.1 Reimplement the Communication Layer

As described in Section 3.2.2, the communication layer of the prototype imposes some severe deficiencies. Thus, a natural next step for *MediaServer* is to redesign and reimplement these classes. Two major considerations need to be kept in mind when redesigning the layer. The first is that process control should not be implemented in this layer, but instead left up to the layer implementing the protocol directly. Second, the message format defined in the protocol should be implemented in the communication layer. The communication layer should be nothing more than a message passing layer for *MediaServer*.

5.2.2 Write Capability

A very important addition to the MediaServer protocol would be semantics for providing write-capability. Two major issues which will need to be addressed are how to best augment the authentication of the MediaServer and how to solve the coherency problems. A higher level issue is whether solutions to these problems should be added to the protocol, or instead left up to implementors. Adding solutions to the protocol imposes restrictions on implementations, while making them implementation issues potentially reduces interoperability of different implementations of the protocol.

5.2.3 World Wide Web Server/Clients

One potential direction for future work is how to make a MediaServer accessible through the World Wide Web. This is a very rich area for research, since it is very

unclear what is the best way to map the object-based MediaServer protocol onto the Web.

The Hypertext Transfer Protocol (HTTP), the protocol on which the Web is based, uses Uniform Resource Locators (URLs) to access elements of the Web. One format which is used heavily on the Web is the "page," a document in which URL's are embedded to form hypertext links.

One possible implementation is to allow users to execute queries using a HTTP "forms" interface, like most Web-based query engines. The HTTP server which handles the form acts as a client to a *MediaServer*. It takes the query returned by the *MediaServer*, extracts the first record, and constructs a page from that record. Any remote objects in the record are "marshalled" such that any data immediately available is placed on the page, and a URL is generated which acts as a OID for the object. Retrieving this URL fully retrieves the rest of the object. A URL for executing "next" on the Cursor is added to the page (in subsequent pages, a URL for "prev" is also added).

Unlike most HTTP servers, which are stateless, this HTTP server must keep references to the generated objects, and their mappings to exported URLs, indefinitely. Since HTTP is a connectionless protocol, there is no reliable way for a client to indicate that it is done with a particular object. As a result, the server must either set an expiration time for each URL (perhaps indicating it on the page on which the URL appears) or limit the number of URL object-references that it stores, using a least-recently-used policy to invalidate old URLs (and free the object they reference) whenever this limit is reached.

This is clearly not the only possible way to allow access to a *MediaServer* from the Web. I do not even claim that it is a good one. It is the very fact that there are so many possible ways to allow access through the Web that makes it a good direction for exploration. Finding the *best* way to do so could take a very large amount of analysis and experimentation.

Appendix A

Client API

This appendix is a reference of the classes included in the client API class library for the *MediaServer*. Each subsection below describes a class within the class library. The classes in this library very closely resemble the classes of the AM2 implementation of the Virtual Database Model. In the future, it is intended that each MediaServer class and each AM2 Database class of the same functional type will each have a copy constructor which takes an object of the other type (for queries and database types).

Each class is described, then the exposed member methods are listed. Properties (data members) of the class cannot be directly accessed. Any properties which a user of the class is allowed to use are exposed through GetName and SetName (if the property is writable), where Name is the name of the property. All method names have a designation prepended to them. This is not a part of the name, but rather indicates whether the method is executed entirely locally (L), remotely on the MediaServer (R), or performs a local action and then calls a remote method (LR).

It should be noted that the implementation of the UTstring class allows "char *" and UTstring to be used interchangeably.

A.1 *MediaServer* command classes

A.1.1 MSmediaserver

MSmediaserver is the main point of connection to a MediaServer. Almost all objects which are created as a result of a connection to a particular server have a pointer to the object of this class which represents the connection to that server.

L:MSmediaserver(const char *ServerName, const int Port)

L:MSmediaserver()

These constructors create a new MSmediaserver, and initialize the internal state. The first sets the name of the server to connect with to ServerName, and the port on which to connect to it to Port. Neither of these constructors actually opens a connection.

R:Open(const char*ServerName, const int Port, const char*Username, const char*Password, Eboolean Queuing, const int timeout, const int QueuePort)

R:Open(const char*Username, const char*Password, Eboolean Queuing, const int timeout, const int QueuePort)

The second form of Open is only validly invoked on an object created by the first constructor. Open creates a connection to the named server on the named port, sending the given username and password for use by the server for authentication. If the Queuing flag is set to eTRUE, then this command will automatically cause the user to be queued by the server if their is no license available. timeout indicates the maximum amount of time to wait for the server to actually grant access, with a value of 0 indicating that the process should wait indefinitely. QueuePort is the port number to wait on if the user is put on the queue. Queuing, timeout, and QueuePort are all optional, and have default values of eFALSE, 0, and the well-known port of the MediaServer (currently 2020), respectively.

R:Close()

Closes the current server connection. This invalidates any and all objects requiring a connection to the server (such as MSobjectRO, MSimageRO, etc.). All temporary files associated with this MediaServer are removed.

L:GetTmpFilePath() *returns* **char***

L:SetTmpFilePath(const char *)

These methods allow access to the path property. All temporary files needed by the MediaServer connection objects are placed in a subdirectory of this directory. The default value for this property is "/usr/tmp".

L:GetErrorCode() *returns* **EclientError**

This method allows read access to the error code property. When any method on a connection object generates an error, this code indicates what kind of error. The possible values are coded in an enum:

```
enum EclientError{
  eSuccess=0,
  eNetworkError,
  eServerError,
  eShutdownNow,
  eIllegalMessage,
  eMSaccessDenied,
  eDBinitError,
  eMSserviceFull,
  eMSnoQueuing,
```

```
eMSqueueFull,
eQueueTimeout,
eDBunknown,
eDBcursorOutOfBounds,
eDBinvalidObjectReference,
eDBobjectNotMedia,
eDBmethodDispatch,
eRmediaUnavailable,
eRmanipulationNotSupported,
eDBnullQuery,
eUnknownError
};
```

L:GetErrorMessage() *returns* UTstring

This method allows read access to the error message property. When any method on a connection object generates an error, this property may contain a description of the error, in a human-readable format.

R:GetAllClasses() *returns* UTlist

Returns a UTlist containing the names of all the classes/tables in the database.

R:GetBaseClasses() *returns* UTlist

Returns a UTlist containing the names of all the classes in the database which have no superclasses.

R:GetSubClasses(const char *Class) *returns* UTlist

Returns a UTlist containing the names of all the subclasses of Class.

R:GetSuperClasses(const char *Class) *returns* UTlist

Returns a UTlist containing the names of all the superclasses of Class.

R:GetAttributes(const char *Class) *returns* UTlist

Returns a UTlist of UTlists. Each sublist contains the name of the type of an attribute of Class and the name of that attribute.

R:GetMethods(const char *Class) *returns* UTlist

Returns a UTlist of UTlists. Each sublist contains the name of a method of Class, the name of the return type for that method, and a UTlist containing the names of the types of the arguments of the method.

R:Execute(const char *StrQuery) *returns* **MScursor**

Sends StrQuery directly to the underlying database for execution and returns the MScursor which contains the results.

LR:Execute(MSquery *Query)

Translates Query into an SQL query string and invokes Execute(const char*).

A.1.2 MSquery

An object of this class is totally independent of any MSmediaserver objects. An application builds up a query using the methods below, then calls MSmediaserver's Execute method with the query as an argument. The same query can be used to execute a query for any MSmediaserver.

The query can be constructed using set_query or the other set_ methods. The first constructs a query as an SQL query string, the other composes it out of lists. In the latter case, an SQL query of the form "select {UTlist} from {UTlist} order by {UTlist}" will be formed.

L:MSquery()

Constructs a null query.

L:MSquery(MSquery)

The copy constructor.

L:set_select_list(UTlist)

Sets the select list for the query.

L:set_from_list(UTlist)

Sets the from list for the query.

L:set_order_by(UTlist)

Sets the order by list for the query.

L:construct_from_list()

Sets the query string to the query represented by the select, from, and order by UTlists.

L:set_query_str(const char*)

Sets the query string directly.

L:get_select_list() *returns* **UTlist**

Gets the select list for the query.

L:get_from_list() *returns* **UTlist**

Gets the from list for the query.

L:get_order_by() *returns* **UTlist**

Gets the order by list for the query.

L:get_query_str() *returns* **const char***

Gets the query string directly.

L:bind(int n, UTvalue Value)

Bind Value to the nth input marker in the query string.

L:bind(UTlist Values)

Bind the elements of Values to the first n input markers in the query string, where n is the number of elements in Values.

A.1.3 MScursor

This class is used to access the results of a query that has been executed on the MediaServer. It provides access to the results without requiring the entire result to be transmitted from the server to the client upon query execution. This delayed binding of the query result to client objects eliminates wasteful overhead upon query execution at the expense of possible small delays upon retrieval of each result.

R:Open(int i) *returns* **UTlist**

Returns the UTlist of data values which represents the result in the ith row of the Cursor. If i is outside the range of the Cursor, an empty list is returned. The first row of a result is position 1.

LR:Next() *returns* **UTlist**

Returns the UTlist of data values which represents the result in the next row of the Cursor. If Next is called on a Cursor positioned at the last row of the result, an empty list is returned. If Next is called on a previously unaccessed Cursor, it returns the first result (position 1).

LR:Prev() *returns* **UTlist**

Returns the UTlist of data values which represents the result in the previous row of the Cursor. If Prev is called on a Cursor positioned at the first row of the result, an empty list is returned.

L:IsLast() *returns* **Eboolean**

Returns eTRUE if the cursor is currently positioned at the last result row.

L:GetPosition() *returns* **int**

Returns the current row position of the Cursor. The first row of a Cursor is position 1.

L:RowCount() *returns* **int**

Returns the number of result rows in the Cursor.

L:ColumnCount() *returns* **int**

Returns the number of columns in each result.

L:GetTypeList() *returns* **UTlist**

Returns a UTlist containing the names of the data types in each result. The first element of this list is the name of the type of the first column, and so on.

L:Success() *returns* **Eboolean**

Returns eTRUE if the Cursor is valid. Returns eFALSE if the query which generated this Cursor did not execute successfully.

A.1.4 MSobjectRO

This class represents any object in the database which cannot be represented using any other Virtual Database Base Type (int, real, string, Boolean, Date, Monetary, Time, Timestamp, Set, or Media). It can be used to access the attributes and methods defined on the object.

L:get_name() *returns* **const char ***

Returns the name of the class to which this object belongs.

R:getAttribute(const char *Aname) *returns* **UTvalue**

Returns the value of the attribute Aname. If Aname is not a valid attribute name, the Unset value is returned.[1]

R:executeMethod(const char *Mname, UTlist Args) *returns* **UTvalue**

Invokes method Mname with arguments Args on this object and returns the resulting value. If Mname is not a valid method name, the Unset value is returned.

A.1.5 MSmediaRO

This class maintains all type independent state information of a media element exported by a server. It provides methods to examine and retrieve the data from the server. MSmediaRO is an abstract base class from which specific media classes, such as MSimagRO below, inherit. In order to simplify the use of media classes, each MSmediaRO object knows the type (image, *etc.*) of the media it references.

R:GetByStream() *returns* **NWxdrStream**

If the media data has not yet been transported, this method sends the sendMedia command to the server, sets up the connection with the Retriever if necessary, and returns the stream over which the media element is being sent. If the element has been transported (*i.e. it is stored locally in a temporary file.*, return an error. If this method returns an error (by returning a NULL string), and the application calls GetErrorCode() on the MSmediaserver object, a code of eSuccess indicates that the element exists locally.

R:GetByFile() *returns* **char ***

If the media element is already stored locally, returns the filename of the temporary file. Otherwise, the method sends the sendMedia command to the server, sets up the connection with the Retriever if necessary, and retrieves the media element into a temporary file. Then it returns the temporary filename.

L:GetType() *returns* **EmediaType**

Returns the type of media contained. The possible values are encoded in an enum:

```
enum EMediaType{
  eUnknownT=0,
  eImage,
```

[1]Actually, in the current version of the Virtual Database classes, when an unknown attribute name is sent, the server crashes. In the future, the procedure will actually signal an exception, a mechanism which allows a function to indicate an error without having to return some type of error-indicating value.

```
eVideo,
eAudio,
eFulltext
};
```

A.1.6 MSimageRO *inherits from* MSmediaRO

This class represents an image media component. MSimageRO inherits all general media functionality from MSmediaRO. It provides methods specifically tailored to working with Images. It is mostly unimplemented currently due to deficiencies in the AM2 database classes.

L:GetImageFormat() *returns* EimageFormat

Returns a a code indicating the image format of the represented image. This function is currently not implemented, since the database classes are currently unable to report the type of an image.

Manipulate() *returns* MSimageRO

Currently unsupported, this method is intended to implement the Manipulate function of the protocol for images. This method overrides the MSmediaRO method of the same name.

A.2 Database Classes

Although the ability to create, set, and alter many of the database objects is exposed through the implementation of the following classes, the read-only nature of the operation of the MediaServer makes doing so virtually useless. For this reason, the descriptions below only list the accessor methods for the given objects, and it is assumed that they are created/set somewhere inside a MediaServer class.

A.2.1 MSdateRO

GetDateStr() *returns* UTstring

Returns the date in the form of a string, such as "13/10/1994".

GetDateList() *returns* UTlist

Returns the date in the form of a UTlist of integers, {month, day , year}.

A.2.2 MSmonetaryRO

GetAmount() *returns* float

Returns the amount of the monetary value.

A.2.3 MStimeRO

GetTimeStr() *returns* **UTstring**

Returns the time in the form of a string, such as "14:03:22".

GetTimeList() *returns* **UTlist**

Returns the time in the form of a UTlist of integers, {hour, minute, second}.

A.2.4 MStimestampRO

GetTstampStr() *returns* **UTstring**

Returns the timestamp in the form of a string, such as "15:04:03 13/10/1994".

A.2.5 MSsetRO

GetList() *returns* **UTlist**

Returns the set in the form of a UTlist of the set members. Thus, the normal UTlist methods, such as Count() and At(n), can be used to determine the size of the set and to retrieve the elements at various points in it.

Bibliography

[1] T Arndt. A survey of recent research in image database management. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 92–97, 1990.

[2] S. Aabo. The Fridtjof Nansen picture database: experiences in building a text/image database. *Microcomputers for Information Management*, 11(1):23–39, 1994.

[3] S. Christodoulakis, D. Anestopoulos, and S. Argyropoulos. Data organization and storage hierarchies in a multimedia server. In *Digest of Papers, COMPCON Spring '93*, pages 596–604, 1993.

[4] M.H. O'Docherty and C.N. Daskalakis. Multimedia information systems: The management and semantic retrieval of all electronic data types. *Computer Journal*, 34(3):225–238, 1991.

[5] The MIT AthenaMuse Consortium. AthenaMuse® 2 Beta 2 Documentation. CECI internal document, April 1995.

[6] The MIT AthenaMuse Consortium. AthenaMuse® 2 Design Specification, Version 1.4. CECI internal document, December 1993.

[7] S. Lerman and S. Centurino. Image server architecture and design specification. CECI internal document, March 1994.

[8] Network Working Group, Sun Microsystems, Inc. XDR: External data representation standard. Request for Comments #1014, June 1987.

[9] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[10] Thomas Keffer. *Tools.h++ Introduction and Reference Manual.* Rogue Wave Software, Inc., 1994.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[12] Stanley B. Lippman. *C++ Primer.* Addison-Wesley Publishing Company, 2nd edition, 1993.

[13] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, 1986.

[14] The MIT AthenaMuse Consortium. AthenaMuse® 2 Interface Specification. CECI internal document, January 1994.

[15] J. Postel. File transfer protocol. Request for Comments #765, June 1980.

[16] W. Richard Stevens. *UNIX Network Programming.* PTR Prentice Hall, 1990.